Final Exam Solutions

Problem 1. True/False [36 points] (18 parts)

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

(a) T F [2 points] Polynomial: good. Exponential: bad.

Solution: True. This is a general rule-of-thumb mentioned in lecture.

(b) T F [2 points] Radix sort runs correctly when using any correct sorting algorithm to sort each digit.

Solution: False. It must use a stable sorting algorithm.

(c) T F [2 points] Given an array $A[1..n]$ of integers, the running time of Counting Sort is polynomial in the input size n.

Solution: False. Counting Sort's running time depends on the size of the numbers in the input, so it is pseudo-polynomial.

(d) T F [2 points] Given an array $A[1..n]$ of integers, the running time of Heap Sort is polynomial in the input size n.

Solution: True. Heap Sort runs in $O(n \log n)$ time on a RAM machine.

(e) T F [2 points] Any n-node unbalanced tree can be balanced using $O(\log n)$ rotations.

Solution: False. The worst-case unbalanced tree is a list, and balancing it requires $\Omega(n)$ rotations.

(f) T F [2 points] If we augment an n-node AVL tree to store the size of every rooted subtree, then in $O(\log n)$ we can solve a *range query*: given two keys x and y, how many keys are in the interval $[x, y]$?

Solution: True. The question describes range trees, as implemented in Problem Set 3.

(g) T F [2 points] AVL trees can be used to implement an optimal comparison-based sorting algorithm.

Solution: True. AVL trees can be used to sort N numbers in $O(N \log N)$ time, by inserting all the numbers in the tree, and iteratively calling NEXT-LARGEST N times.

(h) T F [2 points] Given a connected graph $G = (V, E)$, if a vertex $v \in V$ is visited during level k of a breadth-first search from source vertex $s \in V$, then every path from s to v has length at most k.

Solution: False. The level of a vertex only provides the length of the *shortest* path from s.

(i) T F [2 points] Depth-first search will take $\Theta(V^2)$ time on a graph $G = (V, E)$ represented as an adjacency matrix.

Solution: True. In this case, finding the neighbors of a vertex takes $O(V)$ time, which makes the total running time $\Theta(V^2)$.

(j) T F [2 points] Given an adjacency-list representation of a directed graph $G = (V, E)$, it takes $O(V)$ time to compute the in-degree of every vertex.

Solution: False. The adjacency list structure needs to be traversed to find the incoming edges for each vertex. This structure has total size $\Theta(V + E)$, so this takes $\Theta(V + E)$ time to compute.

(k) T F [2 points] For a dynamic programming algorithm, computing all values in a bottom-up fashion is asymptotically faster than using recursion and memoization.

Solution: False. A bottom-up implementation must go through all of the subproblems and spend the time per subproblem for each. Using recursion and memoization only spends time on the subproblems that it needs. In fact, the reverse may be true: using recursion and memoization may be asymptotically faster than a bottom-up implementation.

(l) T F [2 points] The running time of a dynamic programming algorithm is always $\Theta(P)$ where P is the number of subproblems.

Solution: False. The running time of a dynamic program is the number of subproblems times the time per subproblem. This would only be true if the time per subproblem is $O(1)$.

(m) T F [2 points] When a recurrence relation has a cyclic dependency, it is impossible to use that recurrence relation (unmodified) in a correct dynamic program.

Solution: True. We need to first perform a modification like the one seen in the recitation notes.

(n) T F [2 points] For every dynamic program, we can assign weights to edges in the directed acyclic graph of dependences among subproblems, such that finding a shortest path in this DAG is equivalent to solving the dynamic program.

Solution: False. We saw a counter-example where we couldn't do this in the matrix parenthesization problem.

(o) T F [2 points] Every problem in NP can be solved in exponential time.

Solution: True. NP is contained in EXP.

(p) T F [2 points] If a problem X can be reduced to a known NP-hard problem, then X must be NP-hard.

Solution: False. The reverse, however, is true: if a known NP-hard problem can be reduced to X then X must be NP-hard.

(q) T F [2 points] If P equals NP, then NP equals NP-complete.

Solution: True. A problem X is NP-hard iff any problem in NP can be reduced in polynomial time to X. If P equals NP, then we can reduce any problem in NP to any other problem by just solving the original problem.

(r) T F [2 points] The following problem is in NP: given an integer $n = p \cdot q$, where p and q are N-bit prime numbers, find p or q.

Solution: True. An answer a to the problem can be checked in polynomial time by verifying that $n \mod a = 0$ (and a is not 1 or n). So the factoring problem is in NP. Cryptographic systems (e.g. RSA) often assume that factoring is not in P. False was also accepted because this is not a decision problem.

Problem 2. Sorting Scenarios [9 points] (3 parts)

Circle the number next to the sorting algorithm covered in 6.006 that would be the best (i.e., most efficient) for each scenario in order to reduce the expected running time. You do not need to justify your answer.

(a) [3 points] You are running a library catalog. You know that the books in your collection are almost in sorted ascending order by title, with the exception of one book which is in the wrong place. You want the catalog to be completely sorted in ascending order.

1. Insertion Sort
2. Merge Sort
3. Radix Sort
4. Heap Sort
5. Counting Sort

Solution: Insertion sort will run in $O(n)$ time in this setting.

(b) [3 points] You are working on an embedded device (an ATM) that only has 4KB (4,096 bytes) of free memory, and you wish to sort the 2,000,000 transactions withdrawal history by the amount of money withdrawn (discarding the original order of transactions).

1. Insertion Sort
2. Merge Sort
3. Radix Sort
4. Heap Sort
5. Counting Sort

Solution: Heap sort, because it is in-place.

(c) [3 points] To determine which of your Facebook friends were early adopters, you decide to sort them by their Facebook account ids, which are 64-bit integers. (Recall that you are super popular, so you have very many Facebook friends.)

1. Insertion Sort
2. Merge Sort
3. Radix Sort
4. Heap Sort
5. Counting Sort

Solution: Radix sort

Problem 3. Hotel California [20 points] (5 parts)

You have decided to run off to Los Angeles for the summer and start a new life as a rockstar. However, things aren't going great, so you're consulting for a hotel on the side. This hotel has N one-bed rooms, and guests check in and out throughout the day. When a guest checks in, they ask for a room whose number is in the range $[l, h]$.[1]

You want to implement a data structure that supports the following data operations as efficiently as possible.

1. INIT(N): Initialize the data structure for N empty rooms numbered $1, 2, \ldots, N$, in polynomial time.
2. COUNT(l, h): Return the number of **available** rooms in $[l, h]$, in $O(\log N)$ time.
3. CHECKIN(l, h): In $O(\log N)$ time, return the first empty room in $[l, h]$ and mark it occupied, or return NIL if all the rooms in $[l, h]$ are occupied.
4. CHECKOUT(x): Mark room x as not occupied, in $O(\log N)$ time.

(a) [6 points] Describe the data structure that you will use, and any invariants that your algorithms need to maintain. You may use any data structure that was described in a 6.006 lecture, recitation, or problem set. Don't give algorithms for the operations of your data structure here; write them in parts (b)–(e) below.

Solution: We maintain a range tree, where the nodes store the room numbers of the rooms that are not occupied.

Recall from Problem Set 3 that a range tree is a balanced Binary Search Tree, where each node is augmented with the size of the node's subtree.

[1] Conferences often reserve a contiguous block of rooms, and attendees want to stay next to people with similar interests.

(b) [3 points] Give an algorithm that implements INIT(N). The running time should be polynomial in N.

Solution: All the rooms are initially empty, so all their numbers $(1 \ldots N)$ must be inserted into the range tree.

```
INIT(N)
1  for i ∈ 1...N
2      INSERT(i)
```

(c) [3 points] Give an algorithm that implements COUNT(l, h) in $O(\log N)$ time.

Solution: The COUNT method in range trees returns the desired answer. The number of tree nodes between l and h is exactly the number of unoccupied rooms in the $[l, h]$ interval.

(d) [5 points] Give an algorithm that implements CHECKIN(l, h) in $O(\log N)$ time.

Solution: Finding the first available room with number $\leq l$ is equivalent to finding the successor of $l - 1$ in the BST. The

```
CHECKIN(l, h)
1  r = NEXT-LARGEST(l − 1)
2  if r.key > h
3       return NIL
4  DELETE(r.key)
5  return r.key
```

(e) [3 points] Give an algorithm that implements CHECKOUT(x) in $O(\log N)$ time.

Solution: When a guest checks out of a room, the room becomes unoccupied, so its number must be inserted into the range tree.

```
CHECKOUT(x)
1  INSERT(x)
```

Problem 4. Hashing [15 points] (3 parts)

Suppose you combine open addressing with a limited form of chaining. You build an array with m slots that can store two keys in each slot. Suppose that you have already inserted n keys using the following algorithm:

1. Hash (key, probe number) to one of the m slots.
2. If the slot has less than two keys, insert it there.
3. Otherwise, increment the probe number and go to step 1.

Given the resulting table of n keys, we want to insert another key. We wish to compute the probability that the first probe will successfully insert this key, i.e., the probability that the first probe hits a slot that is either completely empty (no keys stored in it) or half-empty (one key stored in it).

You can make the uniform hashing assumption for all the parts of this question.

(a) [5 points] Assume that there are exactly k slots in the table that are completely full. What is the probability $s(k)$ that the first probe is successful, given that there are exactly k full slots?

Solution: There are $m-k$ possibilities for a successful landing of the first probe out of m total landings. The probability of landing in any slot is $\frac{1}{m}$. Therefore, success probability is $\frac{m-k}{m}$.

(b) [5 points] Assume that $p(k)$ is the probability that there are exactly k slots in the table that are completely full, given that there are already n keys in the table. What is the probability that the first probe is successful in terms of $p(k)$?

Solution:

$$\sum_{k=0}^{\frac{n}{2}} p(k) \cdot \frac{(m-k)}{m}$$

(c) [5 points] Give a formula for $p(0)$ in terms of m and n.

Solution: $p(0)$ is essentially the probability that no keys collide. The probability that the first element doesn't collide with any previous keys is 1. The probability that the second element doesn't collide with any previous keys is $1 - 1/m$. In general, the probability that the ith element doesn't collide with any previous keys, conditioned on the assumption that previous keys did not collide and thus occupy $i-1$ slots, is $1 - (i-1)/m$. Therefore the overall probability is the product

$$\prod_{i=1}^{n} \left(1 - \frac{i-1}{m}\right) = \frac{m!}{(m-n)!} \cdot \frac{1}{m^n}.$$

Problem 5. The Quadratic Method [10 points] (1 parts)

Describe how you can use Newton's method to find a root of $x^2 + 4x + 1 = 0$ to d digits of precision. Either reduce the problem to a problem you have already seen how to solve in lecture or recitation, or give the formula for one step of Newton's method.

Solution: There are two solutions to this problem. The first is the direct application of Newton's method. The second way is to use the formula for the roots of a quadratic equation and compute $\sqrt{3}$ to d digits of precision.

For the first method, we use Newton's formula:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

$f'(x) = 2x + 4$. We need $\lg d$ iterations for convergence.

For the second method, we use the formula to find the roots of a quadratic equation. We get:

$$x = -2 \pm \sqrt{3}$$

and then apply Newton's method to find $\sqrt{3}$ to d digits of precision. We require $\lg d$ iterations.

Problem 6. The Wedding Planner [20 points] (2 parts)

You are planning the seating arrangement for a wedding given a list of guests, V.

(a) [10 points] Suppose you are also given a lookup table T where $T[u]$ for $u \in V$ is a list of guests that u knows. If u knows v, then v knows u. You are required to arrange the seating such that any guest at a table knows every other guest sitting at the same table either directly or through some other guests sitting at the same table. For example, if x knows y, and y knows z, then x, y, z can sit at the same table. Describe an efficient algorithm that, given V and T, returns the minimum number of tables needed to achieve this requirement. Analyze the running time of your algorithm.

Solution: We can construct an undirected graph $G = (V, E)$ with guests as vertices, and an edge between two vertices means the two guests know each other. Table T represents the adjacency lists for the vertices. Two guests can sit at the same table if there is a path between them. If we start from one vertex s and search the graph using breadth-first search (BFS) or depth-first search (DFS), all the guests that are reachable from s can sit at the same table, and additional tables are needed for vertices that are unreachable from s.

Hence, to find the minimum number of tables, we can iterate through $s \in V$. If s is not visited, increment the number of tables needed and call DFS-VISIT(s, T) or BFS(s, T), marking vertices as visited during the traversal. Return the number of tables needed after iterating through all the vertices. This problem is equivalent to finding the number of connected components in the graph. The running time is $\Theta(V + E)$ because every vertex or edge is visted exactly once. Below is the pseudocode.

```
NUM-TABLES(V, T)
1  visited = {}
2  n = 0
3  for s ∈ V
4      if s ∉ visited
5          n = n + 1
6          add s to visited
7          DFS-VISIT(s, T, visitied)
8  return n
```

```
DFS-VISIT(u, T, visitied)
1  for v ∈ T[u]
2      if v ∉ visited
3          add v to visited
4          DFS-VISIT(v, T, visited)
```

(b) [10 points] Now suppose that there are only two tables, and you are given a different lookup table S where $S[u]$ for $u \in V$ is a list of guests who are on bad terms with u. If v is on bad terms with u, then u is on bad terms with v. Your goal is to arrange the seating such that no pair of guests sitting at the same table are on bad terms with each other. Figure 1 below shows two graphs in which we present each guest as a vertex and an edge between two vertices means these two guests are on bad terms with each other. Figure 1(a) is an example where we can achieve the goal by having A, C sitting at one table and B, E, D sitting at another table. Figure 1(b) is an example where we cannot achieve the goal. Describe an efficient algorithm that, given V and S, returns TRUE if you can achieve the goal or FALSE otherwise. Analyze the running time of your algorithm.

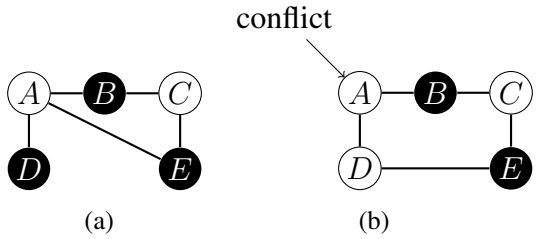

Figure 1: Examples of guest relationships represented as graphs.

Solution: Let $G = (V, E)$ be the undirected graph where V is the set of guests and $(u, v) \in E$ if u and v are on bad terms. S represents the adjacency lists. We can achieve the goal only if there is no cycle with odd length in the graph. We can find out this by iterating through $s \in V$. If s is not visited, color it as WHITE, and call DFS-VISTI(s, S) or BFS(s, S). During the traversal, if v is not visited, mark it as visited and color it BLACK if its parent is WHITE and vice versa. If v is visited, and the color we want to apply is different from its current color, we find a conflict (Figure 1(b)), and we can terminate and return FALSE. If there is no conflict after iterating through all the vertices (Figure 1(a)), return TRUE. The running time is again $O(V + E)$. Below is the pseudocode.

```
CAN-SEPARATE(V, S)
1  color = {}
2  WHITE = 0
3  for s ∈ V
4      if s ∉ color // s is not visited
5          if DFS-VISIT(s, S, WHITE, color) == FALSE
6              return FALSE
7  return TRUE
```

DFS-VISIT$(u, S, \textit{color-to-apply}, \textit{color})$

```
1  if u ∉ color
2       color[u] = color-to-apply
3       for v ∈ S[u]
4            if DFS-VISIT(v, S, 1 − color-to-apply, color) == FALSE
5                 return FALSE
6  else if color[u] ≠ color-to-apply
7            return FALSE
8  return TRUE
```

Problem 7. How Fast Is Your Dynamic Program? [15 points] (5 parts)

In the dynamic programs below, assume the input consists of an integer S and a sequence $x_0, x_1, \ldots, x_{n-1}$ of integers between 0 and S. Assume that each dynamic program uses subproblems (i, X) for $0 \leq i < n$ and $0 \leq X \leq S$ (just like Knapsack). Assume that the goal is to compute $DP(0, S)$, and that the base case is $DP(n, X) = 0$ for all X. **Assume that the dynamic program is a memoized recursive algorithm, so that only needed subproblems get computed.** Circle the number next to the correct running time for each dynamic program.

(a) $$DP(i, X) = \max \left\{ \begin{array}{ll} DP(i+1, X) + x_i, & \\ DP(i+1, X - x_i) + x_i^2 & \text{if } X \geq x_i \end{array} \right\}$$

1. Exponential
2. Polynomial
3. Pseudo-polynomial
4. Infinite

Solution: Pseudo-polynomial

(b) $$DP(i, X) = \max \left\{ \begin{array}{ll} DP(i+1, S) + x_i, & \\ DP(0, X - x_i) + x_i^2 & \text{if } X \geq x_i \end{array} \right\}$$

1. Exponential
2. Polynomial
3. Pseudo-polynomial
4. Infinite

Solution: Infinite

(c) $\boxed{DP(i,X) = \max \left\{ \begin{array}{l} DP(i+1,0) + x_i, \\ DP(0, X - x_i) + x_i^2 \quad \text{if } X \geq x_i \end{array} \right\}}$

1. Exponential
2. Polynomial
3. Pseudo-polynomial
4. Infinite

Solution: Pseudo-polynomial or infinite

(d) $\boxed{DP(i,X) = \max \left\{ \begin{array}{l} DP(i+1,X) + x_i, \\ DP(i+1,0) + x_i^2 \end{array} \right\}}$

1. Exponential
2. Polynomial
3. Pseudo-polynomial
4. Infinite

Solution: Polynomial

(e) $\boxed{DP(i,X) = \max \left\{ \begin{array}{l} DP(i+1, X - \sum S) + (\sum S)^2 \\ \text{for every subset } S \subseteq \{x_0, x_1, \ldots, x_{n-1}\} \end{array} \right\}}$

1. Exponential
2. Polynomial
3. Pseudo-polynomial
4. Infinite

Solution: Exponential

Solution: pseudopolynomial

infinite

pseudopolynomial

polynomial

exponential

Problem 8. Longest Alternating Subsequence [20 points] (6 parts)

Call a sequence $y_1, y_2, \ldots, y_n$ ***alternating*** if every adjacent triple y_i, y_{i+1}, y_{i+2} has either $y_i < y_{i+1} > y_{i+2}$, or $y_i > y_{i+1} < y_{i+2}$. In other words, if the sequence increased between y_i and y_{i+1}, then it should then decrease between y_{i+1} and y_{i+2}, and vice versa.

Our goal is to design a dynamic program that, given a sequence $x_1, x_2, \ldots, x_n$, computes the length of the longest alternating subsequence of $x_1, x_2, \ldots, x_n$. The subproblems we will use are prefixes, augmented with extra information about whether the longest subsequence ends on a descending pair or an ascending pair. In other words, the value $DP(i, b)$ should be the length of the longest alternating subsequence that ends with x_i, and ends in an ascending pair if and only if b is TRUE.

For the purposes of this problem, we define a length-one subsequence to be both increasing and decreasing at the end.

For example, suppose that we have the following sequence:

$$x_1 = 13 \qquad x_2 = 93 \qquad x_3 = 86 \qquad x_4 = 50 \qquad x_5 = 63 \qquad x_6 = 4$$

Then $DP(5, \text{TRUE}) = 4$, because the longest possible alternating sequence ending in x_5 with an increase at the end is is x_1, x_2, x_4, x_5 or x_1, x_3, x_4, x_5. However, $DP(5, \text{FALSE}) = 3$, because if the sequence has to decrease at the end, then x_4 cannot be used.

(a) [4 points] Compute all values of $DP(i, b)$ for the above sequence. Place your answers in the following table:

	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$
$b = \text{TRUE}$						
$b = \text{FALSE}$						

Solution: The following table gives the correct values:

	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$
$b = \text{TRUE}$	1	2	2	2	4	1
$b = \text{FALSE}$	1	1	3	3	3	5

There were several common mistakes on this question. The first mistake was over whether a sequence of length one or two could be alternating. In a length-one or length-two sequence, the number of adjacent triples is zero. As a result, all adjacent triples vacuously satisfy the constraint given. So all length-one and length-two subsequences are alternating.

The second mistake was over the definition of $DP(i, b)$. In the problem, we explicitly define $DP(i, b)$ to be the length of the longest subsequence that ends on x_i and is increasing iff b is TRUE. As a result, $DP(6, \text{TRUE})$ is equal to 1, not 4, because the only ascending subsequence ending on the value $x_6 = 4$ is the subsequence $\langle x_6 \rangle$.

(b) [4 points] Give a recurrence relation to compute $DP(i, b)$.

Solution: The following formula computes $DP(i, b)$ as defined in the problem:

$$DP(i, \text{TRUE}) = 1 + \max_{1 \le j < i \text{ and } x_i > x_j} DP(j, \text{FALSE})$$
$$DP(i, \text{FALSE}) = 1 + \max_{1 \le j < i \text{ and } x_i < x_j} DP(j, \text{TRUE})$$

The most common mistake for this problem involved confusion over the definition of $DP(i, b)$. Many people gave or attempted to give the following recurrence:

$$DP(i, \text{TRUE}) = \max \begin{cases} DP(i-1, \text{TRUE}) \\ DP(i-1, \text{FALSE}) + 1 & \text{if } x_i > x_{i-1} \end{cases}$$
$$DP(i, \text{FALSE}) = \max \begin{cases} DP(i-1, \text{FALSE}) \\ DP(i-1, \text{TRUE}) + 1 & \text{if } x_i < x_{i-1} \end{cases}$$

Unfortunately, this recurrence does *not* compute the value that we asked for. The value $DP(i, b)$ is specifically defined as "the length of the longest alternating subsequence that ends with x_i, and ends in an ascending pair if and only if b is TRUE." The above recurrence relation instead computes the length of the longest alternating subsequence of $x_1, \ldots, x_i$, not necessarily ending on x_i, that ends in an ascending pair if and only if b is TRUE.

(c) [4 points] Give the base cases of your recurrence relation.

Solution: The base cases matching the recurrence relation above are:

$$DP(i, \text{TRUE}) = 1 \qquad \text{if } x_i = \min\{x_1, \ldots, x_i\}$$
$$DP(i, \text{FALSE}) = 1 \qquad \text{if } x_i = \max\{x_1, \ldots, x_i\}$$

(d) [3 points] Give a valid ordering of subproblems for a bottom-up computation.

Solution: The correct order is to iterate through the values of i in increasing order, and compute $DP(i, \text{TRUE})$ and $DP(i, \text{FALSE})$ for each i. The recurrence relation has $DP(i, b)$ dependent only on values $DP(j, \overline{b})$ for $j < i$, so increasing order will give us what we want.

(e) [3 points] If you were given the values of $DP(i, b)$ for all $1 \le i \le n$ and all $b \in \{\text{TRUE}, \text{FALSE}\}$, how could you use those values to compute the length of the longest alternating subsequence of $x_1, x_2, \ldots, x_n$?

Solution: There were multiple acceptable answers here. It's sufficient to either take the maximum of $DP(n, \text{TRUE})$ and $DP(n, \text{FALSE})$, or to take the maximum over all values in the table.

(f) [2 points] When combined, parts **(b)** through **(e)** can be used to write an algorithm such as the following:

```
LONGESTALTERNATINGSUBSEQUENCE(x_1, ..., x_n)
 1  initialize table T
 2  for each subproblem (i, b), in the order given by part (d)
 3      if (i, b) is a base case
 4          use part (c) to compute DP(i, b)
 5      else
 6          use part (b) to compute DP(i, b)
 7
 8      store the computed value of DP(i, b) in the table T
 9
10  use part (e) to find the length of the overall longest subsequence
```

Analyze the running time of this algorithm, given your answers to the questions above.

Solution: Computing the recurrence for $DP(i, b)$ takes time $\Theta(i)$. When we sum this up over the values of i ranging from 1 to n, we get $\Theta(n^2)$ for our running-time. Note, however, that what mattered for this question was correctly analyzing the run-time for the recurrence relation *you* gave, so answers of $O(n^2)$ would be marked wrong (asymptotically loose) if the recurrence relation given actually resulted in a runtime of $\Theta(n)$.

Problem 9. Paren Puzzle [15 points]

Your local school newspaper, *The TEX*, has started publishing puzzles of the following form:

Parenthesize $6 + 0 \cdot 6$ to maximize the outcome.

Wrong answer: $6 + (0 \cdot 6) = 6 + 0 = 6$.
Right answer: $(6 + 0) \cdot 6 = 6 \cdot 6 = 36$.

Parenthesize $0.1 \cdot 0.1 + 0.1$ to maximize the outcome.

Wrong answer: $0.1 \cdot (0.1 + 0.1) = 0.1 \cdot 0.2 = 0.02$.
Right answer: $(0.1 \cdot 0.1) + 0.1 = 0.01 + 0.1 = 0.11$.

To save yourself from tedium, but still impress your friends, you decide to implement an algorithm to solve these puzzles. The input to your algorithm is a sequence $x_0, o_0, x_1, o_1, \ldots, x_{n-1}, o_{n-1}, x_n$ of $n + 1$ real numbers $x_0, x_1, \ldots, x_n$ and n operators $o_0, o_1, \ldots, o_{n-1}$. Each operator o_i is either addition ($+$) or multiplication ($\cdot$). Give a polynomial-time dynamic program for finding the optimal (maximum-outcome) parenthesization of the given expression, and analyze the running time.

Solution: The following dynamic program is the intended "correct" answer, though it ignores a subtle issue detailed below (which only three students identified, and received bonus points for). It is similar to the matrix-multiplication parenthesization dynamic program we saw in lecture, but with a different recurrence.

1. For subproblems, we use substrings $x_i, o_i, \ldots, o_{j-1}, x_j$, for each $0 \le i \le j \le n$. Thus there are $\Theta(n^2)$ subproblems.

2. To solve $DP[i, j]$, we guess which operation o_k is outermost, where $i \le k < j$. There are $j - i = O(n)$ choices for this guess.

3. The resulting recurrence relation is

$$DP[i, j] = \max_{k=i}^{j-1} \Big(DP[i, k] \; o_k \; DP[k + 1, j] \Big).$$

The base cases are

$$DP[i, i] = x_i.$$

The running time per subproblem is $O(n)$.

4. The dynamic program uses either recursion plus memoization, or bottom-up table construction. A suitable acyclic order is by increasing length ℓ of substring, i.e.,

for $\ell = 0, 1, \ldots, n$:
  for $i = 0, 1, \ldots, n - \ell$:
    $j = i + \ell$

5. The value of the original problem is given by $DP[0, n]$. To actually reconstruct the parenthesization, we can remember and follow parent pointers (the argmax in addition to each $\max$). The overall running time is

$$\Theta(n^2) \cdot O(n) = O(n^3).$$

The subtle issue is that this dynamic program assumes that, in order to maximize the sum or product of two numbers, we aim to maximize the two arguments. This assumption is true if the numbers are all nonnegative, as in the examples. If some numbers can be negative, however, then it is not so easy to maximize the product of two numbers. If both of the numbers are negative, so the product is negative, then the goal is to minimize both numbers (i.e., maximizing their absolute values); but if exactly one of the numbers is negative, so the product is negative, then maximization is equivalent to maximizing the negative number and minimizing the positive number (i.e., minimizing their absolute values).

To deal with this issue, we can define two subproblems: $DP_{\max}[i, j]$ is the maximum possible value for the substring $x_i, \ldots, x_j$, as above, while $DP_{\min}[i, j]$ is the minimum possible value for the same substring. Instead of working out which of the two subproblems we need, we can simply guess among the four possibilities, and choose the best. The recurrence relation thus becomes

$$DP_m[i, j] = \min_{k=i}^{j-1} \min_{m_1, m_2 \in \{\max, \min\}} \Big(DP_{m_1}[i, k] \; o_k \; DP_{m_2}[k + 1, j] \Big).$$

The running time remains the same, up to constant factors.

Problem 10. Sorting Fluff [20 points] (5 parts)

In your latest dream, you find yourself in a prison in the sky. In order to be released, you must order N balls of fluff according to their weights. Fluff is really light, so weighing the balls requires great care. Your prison cell has the following instruments:

- A magic balance scale with 3 pans. When given 3 balls of fluff, the scale will point out the ball with the median weight. The scale only works reliably when each pan has exactly 1 ball of fluff in it. Let MEDIAN(x, y, z) be the result of weighing balls x, y and z, which is the ball with the median weight. If MEDIAN$(x, y, z) = y$, that means that either $x < y < z$ or $z < y < x$.
- A high-precision classical balance scale. This scale takes 2 balls of fluff, and points out which ball is lighter; however, because fluff is very light, the scale can only distinguish between the overall lightest and the overall heaviest balls of fluff. Comparing any other balls will not yield reliable results. Let LIGHTEST(a, b) be the result of weighing balls a and b. If a is the lightest ball and b is the heaviest ball, LIGHTEST$(a, b) = a$. Conversely, if a is the heaviest ball and b is the lightest ball, LIGHTEST$(a, b) = b$. Otherwise, LIGHTEST(a, b)'s return value is unreliable.

On the bright side, you can assume that all N balls have different weights. Naturally, you want to sort the balls using as few weighings as possible, so you can escape your dream quickly and wake up before 4:30pm!

To ponder this challenge, you take a nap and enter a second dream within your first dream. In the second dream, a fairy shows you the lightest and the heaviest balls of fluff, but she doesn't tell you which is which.

(a) [2 points] Give a quick example to argue that you cannot use MEDIAN alone to distinguish between the lightest and the heaviest ball, but that LIGHTEST can let you distinguish.

Solution: Suppose we have $N = 2$, so we have 2 balls, a and b. In order to sort them, we need to decide if $a < b$ or $a > b$. We can't even use MEDIAN because we don't have 3 balls.

(b) [4 points] Given l, the lightest ball l pointed out by the fairy, use $O(1)$ calls to MEDIAN to implement LIGHTER(a, b), which returns TRUE if ball a is lighter than ball b, and FALSE otherwise.

Solution:

LIGHTER(a, b)

```
1  if a == l
2      return a
3  if b == l
4      return b
5  if MEDIAN(l, a, b) == a
6      return a
7  else
8      return b
```

After waking up from your second dream and returning to the first dream, you realize that there is no fairy. Solve the problem parts below without the information that the fairy would have given you.

(c) [6 points] Give an algorithm that uses $O(N)$ calls to MEDIAN to find the heaviest and lightest balls of fluff, without identifying which is the heaviest and which is the lightest.

Solution: The pseudo-code below starts out by weighing the first 3 balls, and repeatedly replaces the median with a new ball, until the balls runs out. The two remaining balls must be the extremes, because an extreme will never be a median, and therefore will never be eliminated.

```
EXTREMES(b, N)
1  x, y = b_1, b_2
2  for i ∈ 3...N
3      z = b_i
4      m = MEDIAN(x, y, z)
       // Set x and y to non-median balls
5      if x == m
6          x, y = y, z
7      if y == m
8          y = z
9  return (x, y)
```

It is not sufficient to call MEDIAN on all 3 groups of adjacent balls and hope that it will rule out all the balls except for the two extremes. Example: given 4 balls with weights 3471, MEDIAN would point at the 2nd ball twice.

(d) [2 points] Explain how the previous parts should be put together to sort the N balls of fluff using $O(N \log N)$ calls to MEDIAN and $O(1)$ calls to LIGHTEST.

Solution: Call EXTREMES (the answer to part c) to obtain the lightest and heaviest balls, then call LIGHTEST to obtain the lightest ball. Last, use LIGHTER (the answer to part b) as the comparison operator in a fast ($O(N \log N)$ time) comparison-based sorting algorithm.

Out of the algorithms taught in 6.006, insertion sort with binary search makes the fewest comparisons. Other acceptable answers are merge-sort and heap-sort, as they all use $O(N \log N)$ comparisons.

(e) [6 points] Argue that you need at least $\Omega(N \log N)$ calls to MEDIAN to sort the N fluff balls.

Solution: The argument below closely follows the proof of the $\Omega(N \log N)$ lower bound for comparison-based sorting.

A call to MEDIAN has 3 possible outcomes, so a decision tree based on MEDIAN calls would have a branching factor of 3. There are $N!$ possible ball permutations, so the decision tree needs $\Omega(\log_3 N!) = \Omega(\log N!) = \Omega(N \log N)$ levels to cover all possible $N!$ permutations.

LOWEST only provides useful information if it is called once, and it reduces the possible permutations to $\frac{N!}{2}$. This doesn't change the result above, because the constant factor gets absorbed by the asymptotic notation.

The lower bound obtained from comparison-based sorting cannot be used without argument, because it is not obvious that this problem is harder than comparison-based sorting. To use this bound correctly, a solution would have to prove that comparison-based sorting can be reduced to this problem, by implementing MEDIAN and LIGHTEST with $O(1)$ comparisons each.

6.006 Introduction to Algorithms
Fall 2011