Introduction to Algorithms
Massachusetts Institute of Technology
Professors Erik Demaine and Srini Devadas

December 16, 2011
6.006 Fall 2011
Final Exam

Final Exam

- 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 180 minutes to earn 180 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.
- This quiz is closed book. You may use **three** $8\frac{1}{2}'' \times 11''$ or A4 or 6.006 cushions as crib sheets (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 writing an algorithm, a *clear* description in English will suffice. Pseudo-code is not required.**
- When asked for an algorithm, your algorithm should have the time complexity specified in the problem with a correct analysis. 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		Problem	Parts	Points	Grade	Grader
1	18	36				6	2	20		
2	3	9				7	5	15		
3	5	20				8	6	20		
4	3	15				9	1	15		
5	1	10				10	5	20		
						Total		180		

Name: _____

Wed/Fri Recitation:	Ying	Kevin	Sarah	Yafim	Victor
	10, 11 AM	**11 AM**	**12, 1 PM**	**12 PM**	**2, 3 PM**

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.

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

(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.

(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.

(e) T (F) [2 points] Any n-node unbalanced tree can be balanced using $O(\log n)$ 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 $[x, y]$?

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

(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.

(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.

(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.

(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.

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

(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.

(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.

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

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

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

(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.

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

(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

(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

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.

1-D Range Tree.

Nodes define themselves by room numbers. If a room is in Tree, it's available. Accessing room numbers with node.key.

Range queries was subject to Problem Set 3. Range Tree is similar to an AVL Tree and a Red Black Tree.

[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.

All rooms are initially empty, we will just insert all room numbers.

```
INIT(N)
1. for room number in {1....N}
2.     INSERT(room number)
```

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

Range Trees already has the method. We can just call COUNT(l,h) for [l,h] interval.

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

```
CHECKIN(l,h)
1. node = NEXT-LARGEST(l-1)
2. if node.key > h do
3.     return None
4. endif
5. DELETE(node)
6. return node.key
```

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

```
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?

$1 - \frac{k}{m} = \boxed{\frac{m-k}{m}}$

(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 $p(k)$?

Previously we assumed k slots are completely full. In this case we are not sure and just have the probability function, which makes first probe more likely to succeed.

$\boxed{p(k) \cdot \frac{m-k}{m}}$

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

$\boxed{\prod_{k=0}^{n-1} \frac{m-k}{m}}$

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.

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

At d steps, x_0 to x_d as you select x_0 initially, result will be x_d, every iteration increases precision by 1 digit.

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.

MAIN ALGORITHM

```
1 visited = empty set
2 reservations = hashtable
3 table_numbers = 0

4 for every guest in V
5 if guest not in visited do
6 add guest to visited
7 table_numbers += 1
8 reservations[guest] = table_numbers
9 INIT_TABLE(guest)
```

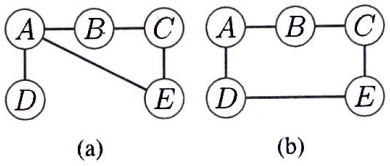
INIT_TABLE(guest)

```
1 for every friend in T[guest]
2 if friend not in visited do
3 add friend to visited
4 reservations[friend] = table_numbers
```

Every guest maximum of 2 times encountered, main algorithm and subroutine INIT_TABLE. Which means algorithm is polynomial.

O(size of V)

(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.



(a) (b)

Figure 1: Examples of guest relationships presented as graphs.

```
1  Set hashtable R
2  Initialize R[g] = 0 for all g ∈ V

3  for every guest in V do
4     if R[guest] equals 0 set R[guest] = 1


5     for every person in S[guest] do
6        if R[guest] equal R[person] return FALSE
7        if R[person] equal 0 do
8           if R[guest] equal 1 set R[person] = 2
9           else set R[person] = 1
10       end if
11    endfor
12 endfor
13 return TRUE
```

Algorithm visits every vertex once and every edge twice at the worst case, time complexity is $O(V+E)$.

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

(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

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

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

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

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

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

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

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:

	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$
$b =$ TRUE	1	2	2	2	4	1
$b =$ FALSE	1	1	3	3	3	5

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

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

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

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

$$DP(i, \text{TRUE}) = 1 \Leftarrow x_i = \min\{x_1, \ldots, x_i\}$$

$$DP(i, \text{FALSE}) = 1 \Leftarrow x_i = \max\{x_1, \ldots, x_i\}$$

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

Increasing order, $j = 1$ to i, memoize $DP(i, \text{TRUE})$ and $DP(i, \text{FALSE})$.

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

Take maximum value in all given values.

(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.

for every position 1 to n, we compute maximum and minimum between $x_0, \ldots, x_i$ which takes $O(n)$ time. Therefore $O(n^2)$.

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.	Parenthesize $0.1 \cdot 0.1 + 0.1$ to maximize the outcome.
Wrong answer: $6 + (0 \cdot 6) = 6 + 0 = 6$.	*Wrong answer:* $0.1 \cdot (0.1 + 0.1) = 0.1 \cdot 0.2 = 0.02$.
Right answer: $(6 + 0) \cdot 6 = 6 \cdot 6 = 36$.	*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.

1- Subproblems: Use substring notation $O[i \leq j \leq n$ as $x_i, o_i, \ldots o_{j-1}, x_j$

Substring $\rightarrow \Theta(n^2)$ subproblems

2- Choices for subproblems: o_k is the operator between two parenthesizations in interval $[i, j]$. k is in $i \leq k < j$ equation.

Therefore choices are $j - i = O(n)$

3- Recurrence Relation: $DP[i,j] = \max_{k=i}^{j-1} (DP[i,k]\, o_k\, DP[k+1, j])$

and base case $DP[i,i] = x_i$

Therefore running time per subproblem is $O(n)$

4- Recurse and memoize or build DP table bottom-up

Topological order $\rightarrow$ increasing substring size - total time $O(n^3)$

5- Original Problem

Result is $DP[0, n]$. You can use parent pointers to recover parens.

Time complexity of DP is $O(n^3)$.

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.

MEDIAN(x,y,z) returns only the middle value. Therefore we don't know anything about other two values.

(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.

```
LIGHTER(a,b)
1. if   a == MEDIAN(l,a,b)  do
2.      return TRUE
3. else
4.      return FALSE
5. endif
```

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.

Basic idea is, lightest and heaviest ball never be the median. If we iterate over all the balls from the start and continue with non-median balls. At the end, remaining two balls are what we are looking for.

$\{b_1 \cdots b_N\}$ and N are inputs.

```
FIND_EXTREMAS(b, N)
1. x, y = b1, b2
2. for i = 3 to N do
3.    m = MEDIAN(x,y,bi)
4.    if  m is x do  x=m
5.    else if  m is y do  y=m
6. endfor
7. return (x,y)
```

(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.

Invoke FIND-EXTREMAS and find lightest ball with LIGHTEST to use comparison function LIGHTER.

Use LIGHTER, sort balls with Merge Sort or Heap Sort algorithms.

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

lower bound

We need to use comparison function at least $\Omega(N \log N)$, and comparison function invokes MEDIAN once every time it's executed. for most efficient algorithms we need to use at least $N \log N$ comparisons, therefore we call MEDIAN at least $N \log N$ times.