Final Exam

Sunday, November 25, 2018 8:50 PM

6p006_Final

82%

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	148	

Name: _____ 1

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

82%

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.

Stable is required

(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.

its linear in size with a pseudopolynomial factor depending on the range of the values

(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.

Trick Question, you bastard!

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

should've paid as you went you bastard!

(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 $[x, y]$?

yea we can climb up and down and tally what we need to from size

(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.

z

The implication here is adjacency list just represents edges with tail at a vertices. To count in-degrees, we need to use a dictionary, or otherwise, and tally em up.

(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.

Given adj list represented using separate arrays for in-deg, of edge

(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.

(n) This is an interesting distinction. DAG form will work only if we're adding a bunch of stuff

or doing a product series and we can take a log. more "intricate" recursive dependencies will not lend itself to min path weight paradigm

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

Thanks smart math guy. You da BeST!

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

NP-hard: problem that's at least as hard as the hardest problem in NP type

NP-complete: problem that is in NP-hard and NP (max of NP)

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

? NP-hard vs NP-complete

if P=NP every NP problem is poly-solvable.

explain why P=NP=NPC

(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

Insertion sort: in-place, stable, O(n^2), optimal substructure sorting algorithm. May be optimal for *nearly* sorted arrays

- From the BC of empty array, it sorts progressively larger slices of the array
- Assuming the i-1 elements are sorted, it places the i'th element in it's correct position
  - This allows the break condition that if the i-1 element obeys the sort property
    - it is implied, since i-2, i-3, ... 0'th element are all smaller than i-1, so we can leave it alone

(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

O(bn)

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]

range query, checkin

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.

range query

Find

(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.

DS: Modified AVL tree

↳ n.size attribute will be changed to n.avail size AND n.occupied size

↳ include n.avail Size field of each node in AVL tree

invariant: X.availSize always matches # of available rooms in subtree rooted at X

[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.

def init(N):
self.
populate a basic AVL tree,
including additional field
n.availSize, which initializes n/
same value n size

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

Find node l, get. availSize of l.left → a
Find node h, get availSize of r.right → b
return tree.root.availSize - a - b

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

even if you had an answer, you'd have worst case O(n) time because all rooms up to h might be checked out.

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

Starting at tree root, descend tree until node with key == x is found. (if node.key < x, descend right, if node.key > x, descend left)

Once node corresponding to room is found, increment node.availSize by 1.

repeat: node = node.parent until: node == NIL

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?

$P(\text{success on } 1^{st} \text{ try}) = \frac{m-k}{m}$

Given uniform hashing

(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 $p(k)$?

$P(1^{st} \text{ probe Success}) = \sum_{k=0}^{\lfloor n/2 \rfloor} p(k)\left(\frac{m-k}{m}\right)$

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

-2

$p(0) = Pr(\text{no collisions occur in } n \text{ inserts})$

Chain rule: $p(0) = \prod_{i=1}^{n} Pr(\text{no collision on } i\text{'th insert} \mid \text{no prior collisions})$

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

Reduce Product Series to a single formula:
PI(from i=1 to n, (m-(i-1))
(can factor out prod series)
= PI(1/m)PI(m-i+1)
(first series term is independent of arg) (second series has short hand form of m! / (m-n)!
= 1/m^n * m!/(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.

Newton's method is a root finding method.

Let $f(x) = x^2 + 4x + 1$

when $f(x) = 0$, x is a root, by defn.

$\frac{\Delta y}{\Delta x} = f(x)$

$\Delta x = \frac{f(x)}{\Delta y}$

-5

Newton's method
has quadratic convergence
quadratic convergence implies "correct digits" are doubling
therefore,
for d digits
it takes lg(d) doublings
lg(d) iterations!

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

$f'(x) = 2x + 4$

$\frac{f(x)}{\Delta x} = f(x)$

Δx

$x_{i+1} = x_i - \frac{2x_i + 4}{x_i^2 + 4x_i + 1}$

Let $x_0 =$ [illegible]

$x_1 =$ [illegible]

$x_2 =$

$x_{i+1} = x_i - \frac{x_i^2 + 4x_i + 1}{2x_i + 4}$

let $x_0 = 0$

$x_1 = -0.25$

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.

undirected graph: Vertices V
edges represent w/ adjacency lists in T

Constraint: every table subgraph must be connected.

PC: Run DFS and count # of connected components

```
main
Tables = 0
Seated = {}
For v in V:
  if ! Seated.hasKey(v):
    Seated[v] = true
    Tables += 1
    FillTable(v, T, Seated)
```

```
Fill Table
For x in T(v):
  if ! Seated.hasKe
    Seated[x] = t
    FillTable(x, T
```

Complexity: O(V + E)

Amortized Analysis, nodes + edges are explored only once

(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.

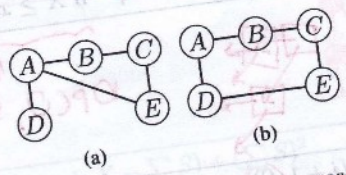


Figure 1: Examples of guest relationships presented as graphs.

- CSP,
- V variables, w/ domain size 2 for all variables
- invariant: ~~any cycle~~ if ∃ cycle in G w/ odd # of unique vertices, the problem is intractable,

PC; Run DFS while tracking depth.
If a back-edge is encountered and depth is odd, return false.
If DFS terminates, return true

Running Time: O(V+E)
(same as DFS)

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

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

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

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

What is the run-time with a cyclic dependency?

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

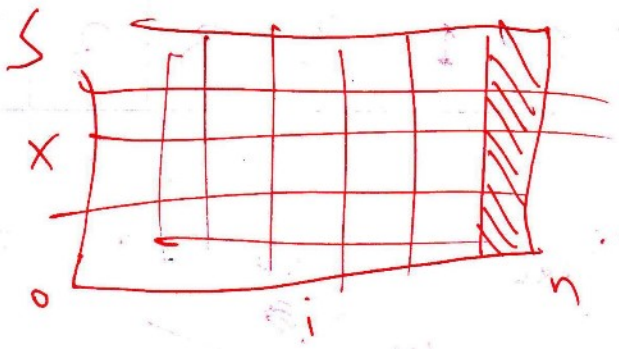
1. Exponential
2. Polynomial
3. Pseudo-polynomial
4. Infinite

(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:

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

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

$$DP(i,b) = \begin{cases} 1 & \text{if } b \text{ and } (x_i > x_{i-j}) \text{ xor } \bar{b} \\ 1 + DP(i-j, \bar{b}) & \text{if } \forall j \in \{1, \ldots, i-1\} \\ & \text{else} \end{cases}$$

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

$DP(1, b) = 1$

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

For i=1 to n:
  For b in {true, false}:
    DP

(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 $x_1, x_2, \ldots, x_n$?

(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.

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.

Decision: choose optimal parenthesization among...

Enumerate: n! choices

1) Subproblems: use prefix and suffix, DP(i,j) $O(n^2)$

2) Guess: where to put next operators

3) Cost per subproblem: $O(n)$

4) Recursion: $DP(i,j) = \max\{O_k(DP(i,k), DP(k+1,j))\ \forall k | i \le k < j\}$ if $j = i+1$, $O_i(x_i, x_j)$

5) Extraction:

DP[

BC:

run time: $O(n^2) \cdot O(n) = O(n^3)$

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 $\text{MEDIAN}(x, y, z)$ be the result of weighing balls x, y and z, which is the ball with the median weight. If $\text{MEDIAN}(x, y, z) = y$, 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 $\text{LIGHTEST}(a, b)$ be the result of weighing balls a and b. If a is the lightest ball and b is the heaviest ball, $\text{LIGHTEST}(a, b) = a$. Conversely, if a is the heaviest ball and b is the lightest ball, $\text{LIGHTEST}(a, b) = b$. Otherwise, $\text{LIGHTEST}(a, b)$'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.

Let x_1, x_2 be balls shown by faerie.

Let x_3 be a third ball

Median will NEVER return a or b

↳ ∴ median will never distinguish between a or b

lightest(a, b) return a, the lightest

(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.

Let c be any other

lighterPointer = median(a, b, c)
if a == lighterPoint:
return true
else:
return False

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.

$\binom{N}{3}$ unique median calls

while |X| > 2:
bye = median(x_1, x_2, x_3)
X.remove(bye)

(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.

-2

```
(a,b) = partc(X)
if partd(a,b):
    lightest = a
else:
    lightest = b
def compareReturnLower(x,y):
    return (a, x, y)
compareSort(X, comparator=)
```

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

Give an argument for why sorting using a median function that returns which of 3 elements is the median will take at LEAST NlogN time in the worst case?

Algorithm must distinguish between n! potential solutions without apriori knowledge

Information received from median reduces uncertainty of, there were once 3 possible outcomes, and now we know which one.

decision tree has branching factor of 3 => NlogN heighty to spread out in optimally efficient way.