Problem 1 (15 pts). *Suppose you are choosing between the following three algorithms, all of which have O(1) base cases for size 1:*

1. *Algorithm A solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.*
2. *Algorithm B solves problems of size n by recursively solving one subproblem of size $n/2$, one subproblem of size $2n/3$, and one subproblem of size $3n/4$ and then combining the solutions in linear time.*
3. *Algorithm C solves problems of size n by dividing them into nine subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.*

What are the running times of each of these algorithms (in asymptotic notation) and which would you choose? You may use the Master Method. Hint: Approach Algorithm B via the substitution method (pp. 211-217) to avoid tough summations. Also, solving that one as $O(n^d)$ for the smallest valid integer d is all I'm looking for.

Solution. By master method: ① five subproblems, half of size; combine linearly.

$a=5, b=2$ $\quad T(n)=5T(\frac{n}{2})+f(n^k)$; $k=1$, $\log_b a = \log_2 5 > k$.

then complexity is: $\Theta(\log_b a) = \Theta(n^{\log_2 5}) = O(n^{2.32})$

② B: 3 subproblem, $n/2$, $2n/3$, $3n/4$ combine linearly.

$T(n) = T(\frac{n}{2}) + T(\frac{2n}{3}) + T(\frac{3n}{4}) + O(n)$.

a). By O notation; Assume the lower bound of $T(n)$ is O^3; then $c(n)^3 \geq c(\frac{n}{2})^3 + c(\frac{2}{3}n)^3 + c(\frac{3}{4}n)^3 + c(n) > 0.84n^3$

then Assume that $T(n) \leq cn^2$, then exist c and m

that let $T(m) \leq cm^2$ and when $n >= m$, let $T(n) \leq cn^2$ (n)

$T(m) = cm^2/4 + 4cm^2/9 + 9cm^2/16 + m$

$= cm^2(\frac{1}{4} + \frac{4}{9} + \frac{9}{16}) + m$

$\approx 1.257\, c\, m^2 + m$

therefore $T(n) = O(n^2)$

③ C: $a=9$, $b=3$, combine in $O(n^2)$ time.

$T(n) = 9T(\frac{n}{2}) + f(n^2)$

$\log_b a = \log_3 9 = 2 = k = 2$, $p=0$, $\Theta(n \cdot \log n)$.

Thus, problem c has the best running time.

Problem 2 (10 pts). *We are given a $n \times n$ board with one square colored black, all others colored white. Your task is to cover the board with "L" shaped tiles, such that all white squares are covered, the black square remains uncovered and there is no overlap among the L-shapes. An L-shape consists of three out of the 4 squares in a 2×2 square (thus one of the $2 \times 2 = 4$ squares is left uncovered by the shape). Design a divide-and-conquer algorithm to find a tiling of the board. Your algorithm will take as input the size n of one side(!) board and the position of the black square. You may assume that $n \geq 2$ is a power of 2. Prove the correctness of your algorithm and analyze its running time. For full credit you must write the recurrence for the running time in your analysis.*

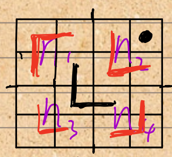
complexity divide original

Tiled (n): $T(n) = 4\left(\frac{n}{2}\right) + O(1)$

{ if n=2:
anyone of the square in 2x2 square is block black.
the left three square is a "L" shape, which fit "L"
else: let n_1, n_2, n_3, n_4 is each one out of 4 square
that is cut by the middle line of opposite sides
the black square must on any one of $n_1 n_2 n_3 n_4$ square.
draw a "L" in the middle of original square but not
in the current square where the black located.
Tiled (n_1)
Tiled (n_2)
Tiled (n_3)
Tiled (n_4)
Then assume that the 3 squares "L" covered is the
black covered square on each of their square
that with a "L" shape for lette "L".

square into four squre. by
master theory: a=4. the input
n is the length of each side.
a=2, when tiled the square is
constant time.

thus. $T(n) = 4\left(\frac{n}{2}\right) + O(1)$ so. overall
take $O(n)$.



graph 2

Solution and correctness: Assume $n = 2^k$ by 2^k square can not be tiled by any one black square and "L" shape that take 3 out of 4 squares for each "L". As graph 2, our algorithm assume the black square on one of the 4 squares that is divided the n square into 4 by the middle point of each side. then we draw one "L" in the middle of square n, but not located in the one of the 4 square that the black square located.
As we divide n square into 4 squares, n_1, n_2, n_3, n_4, each of the squares has a black square that is from the letter "L" and original black square. repeat this process on square of n_1 n_2 n_3 n_4 as square n, all square must automaticlly left a "L" shape.

proof: By contradiction: Assume our algorithm is not tiled the $n \times n$ square. then there must be not a "L" shape left after a black square located. by our algorithm. after divide n into 4 small square, there must not be a "L" blank shape left. repeatly. when each of 4 square is divided into 2x2 square and one black square already exist. it must be a "L" shape left. That is conflict with the assumption.

Problem 3. *(15 pts) A bank has n bank cards which each belong to a bank account. Some cards may belong to the same bank account. It is infeasible to read the account information directly from the cards, but the bank provides a machine which can test whether two cards are registered to the same account.*

a:*(5 pts) Design an algorithm which determines whether there is a set of more than $\frac{n}{2}$ cards which belong to a single account. At worst it should use the machine $O(n \log n)$ times. Prove it is correct.*

same. → one
diff → discard

ferinmict correct

b:*[10 pts] Now we'll try to solve it faster. Consider the following procedure when n is even: Arbitrarily pair up the cards. Test each pair and if the two elements are different, discard both of them; if they are the same, keep just one of them. Show that this procedure preserves the majority element: if the original batch of cards had a majority element, then the reduced batch of cards has the same majority element. Use this procedure to build a divide-and-conquer algorithm for all n. Analyze its running time by writing a recurrence relation and evaluating it.*

a) Solution: By observation, if there is a majority element in more than $n/2$ locations. if we split the array into two halves, the same elements will be a majority in at least one of the two halves. Then if an element is a majority in one of the two sides, then it is not necessarity to the whole array. Thus, when a a majority element is returned from one of the two halves, we have to compare it against all other elements.

s = length of number of cards in array.
l = first half size of array
r = left size of array.
m = middle index of array of cards.

Algorithm: Majority (S) {

if n=2
  if card1 = card2 : return card1
  else return nothing.
let S_1 = set of first $\frac{n}{2}$ elements,
let S_2 = set of remindy $n/2$ elements.
Majority (S_1)
majority (S_2)
if an element is returned.
  then test this element against all other elements.
if no element with majority has been found.
  Majority (S_2)
  if an element is returned
    then test this element against all other elements
Return a majority element if one is found.

complexity: Divide and conquer.
$T(n) = 2(\frac{n}{2}) + O(n)$ by master theory: $O(n \log n)$

correctness: If there is a majority element in the array, then one of the two recursive calls, will return this element, and by comparing it with the remining elements of the array we can make sure that it is indeed the majority element.

B). n ⇒ the length of number of cards in array.

find majority (n)
if n=2
  element1 = element2.
  return element1
else return nothing.
n_1 = the first half of cards in array
n_2 = the second hat of cards in array.
find majority (n_1)
find majority (n_2)
if n_1 != null;
  return element in n_1,
else return element in n_2.

complexity: $T(n) = 2(\frac{n}{2}) + O(1)$.
as it cut of all the distinguish pair card, therefor. a = 1.
the running time is O(n).

proof: Contradiction. Assume this procucure didnot preserve the majority elements. By divide & conquer, after arbitrarily pair up 2 numbers recursivly. when compair them recursivelly. discard both if they are distinguish, otherwise, keep one. Therefore, by pigeon hole principle. the batch of majority number must at least two of them are same. in one pair that will be return. Thus. Assume is not correct.

4). Suppose you're managing a consulting team of expert computer hackers, and each week you have to choose a job for them to undertake. Now, as you can well imagine, the set of possible jobs is divided into those that are *low-stress* (e.g., setting up a Web site for a class at the local elementary school) and those that are *high-stress* (e.g., protecting the nation's most valuable secrets, or helping a desperate group of Cornell students finish a project that has something to do with compilers). The basic question, each week, is whether to take on a low-stress job or a high-stress job.

If you select a low-stress job for your team in week i, then you get a revenue of $\ell_i > 0$ dollars; if you select a high-stress job, you get a revenue of $h_i > 0$ dollars. The catch, however, is that in order for the team to take on a high-stress job in week i, it's required that they do no job (of either type) in week $i-1$; they need a full week of prep time to get ready for the crushing stress level. On the other hand, it's okay for them to take a low-stress job in week i even if they have done a job (of either type) in week $i-1$.

So, given a sequence of n weeks, a *plan* is specified by a choice of "low-stress," "high-stress," or "none" for each of the n weeks, with the property that if "high-stress" is chosen for week $i > 1$, then "none" has to be chosen for week $i-1$. (It's okay to choose a high-stress job in week 1.) The *value* of the plan is determined in the natural way: for each i, you add ℓ_i to the value if you choose "low-stress" in week i, and you add h_i to the value if you choose "high-stress" in week i. (You add 0 if you choose "none" in week i.)

The problem. Given sets of values $\ell_1, \ell_2, \ldots, \ell_n$ and $h_1, h_2, \ldots, h_n$, find a plan of maximum value. (Such a plan will be called *optimal*.)

Example. Suppose $n = 4$, and the values of ℓ_i and h_i are given by the following table. Then the plan of maximum value would be to choose "none" in week 1, a high-stress job in week 2, and low-stress jobs in weeks 3 and 4. The value of this plan would be $0 + 50 + 10 + 10 = 70$.

	Week 1	Week 2	Week 3	Week 4
ℓ	10	1	10	10
h	5	50	5	1

$\ell_i = 20 + 50$
$= 70$

Algorithm:
i → the length of total of weeks

(a) Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer.

```
For iterations i = 1 to n
  If h_{i+1} > ℓ_i + ℓ_{i+1} then
    Output "Choose no job in week i"
    Output "Choose a high-stress job in week i+1"
    Continue with iteration i+2
  Else
    Output "Choose a low-stress job in week i"
    Continue with iteration i+1
  Endif
End
```

To avoid problems with overflowing array bounds, we define $h_i = \ell_i = 0$ when $i > n$.

In your example, say what the correct answer is and also what the above algorithm finds.

(b) Give an efficient algorithm that takes values for $\ell_1, \ell_2, \ldots, \ell_n$ and $h_1, h_2, \ldots, h_n$ and returns the *value* of an optimal plan.

solution:

a). By the given algorithm. that is greedy. suppose the h_{i+2} day, that has very large value, however, after the algorithm pick day h_{i+1}, cannot get day h_{i+2} anymore. namely since last iteration was not 'none' high-stree job will be not chosen.

eg.

	week 1	week 2	week 3	week 4
ℓ	2	2	2	2
h	1	5	20	10

here $h_2(5) > \ell_1 + \ell_2(2+2)$
chose h_{i+2} by their algorithm, However. $h_3 > h_2$. but we cannot pick h_3. as h_2 already picked.

if seq(0) return 0;
if seq(1) return max(h_1, ℓ_1)
else return
max{(opT(i-2) + hi), (opT(i-1) + ℓ_i)}
memiozation. keep in array. the previous value
(0, max(ℓ_1, h_1)
opT(0), opT(1) opT(2) ··· opT(i).

claim: when considering week j for opT(j), we cover all possible solutions on job 1, 2 ··· j.
proof; by Induction: Base case, when the sequence of week is 0, the maxmum is 0. or the sequence week is 1. we pick the maxmum one between ℓ_1 and h_1.
Inductive step; Assume, given a sequence of (i-2) weeks. we found a maxmum value. how we prove the sequence of (i+1) week. by our algorithm, when we pick h_i, we skip one day before i. we find the value of day(i) plus value before h(i-2) day. Similarly, we sum the value of i day ℓ. and day (i-1) of value ℓ. Because we already assume the maxmum value before (i-2) day are maxmum we got. we add the maxmum value of either h or i. the final set of value also maximum.

9. You're helping to run a high-performance computing system capable of processing several terabytes of data per day. For each of n days, you're presented with a quantity of data; on day i, you're presented with x_i terabytes. For each terabyte you process, you receive a fixed revenue, but any unprocessed data becomes unavailable at the end of the day (i.e., you can't work on it in any future day).

You can't always process everything each day because you're constrained by the capabilities of your computing system, which can only process a fixed number of terabytes in a given day. In fact, it's running some one-of-a-kind software that, while very sophisticated, is not totally reliable, and so the amount of data you can process goes down with each day that passes since the most recent reboot of the system. On the first day after a reboot, you can process s_1 terabytes, on the second day after a reboot, you can process s_2 terabytes, and so on, up to s_n; we assume $s_1 > s_2 > s_3 > \cdots > s_n > 0$. (Of course, on day i you can only process up to x_i terabytes, regardless of how fast your system is.) To get the system back to peak performance, you can choose to reboot it; but on any day you choose to reboot the system, you can't process any data at all.

The problem. Given the amounts of available data $x_1, x_2, \ldots, x_n$ for the next n days, and given the profile of your system as expressed by $s_1, s_2, \ldots, s_n$ (and starting from a freshly rebooted system on day 1), choose the days on which you're going to reboot so as to maximize the total amount of data you process.

Example. Suppose $n = 4$, and the values of x_i and s_i are given by the following table.

day of reboot cannot be chosen

	Day 1	Day 2	Day 3	Day 4
x	10	1	7	7
s	8	4	2	1

The best solution would be to reboot on day 2 only; this way, you process 8 terabytes on day 1, then 0 on day 2, then 7 on day 3, then 4 on day 4, for a total of 19. (Note that if you didn't reboot at all, you'd process $8 + 1 + 2 + 1 = 12$; and other rebooting strategies give you less than 19 as well.)

(a) Give an example of an instance with the following properties.
- There is a "surplus" of data in the sense that $x_i > s_1$ for every i.
- The optimal solution reboots the system at least twice.

In addition to the example, you should say what the optimal solution is. You do not need to provide a proof that it is optimal.

(b) Give an efficient algorithm that takes values for $x_1, x_2, \ldots, x_n$ and $s_1, s_2, \ldots, s_n$ and returns the total *number* of terabytes processed by an optimal solution.

Solution: a)

day	1	Day 2	Day 3	Day 4	Day 5	Day 6
X	4	5	1	100	8	300
S	3	6	2	200	9	600

b).

complexity: $O(n \cdot m)$

```
for i = 1 to n:
    for j = i to n:
        f(di, li) = max of min(Xi, Si) + f(di+1, li+1), f(dj+1, 1)
```

minimization the max value.

min(Xi, Si) + f(di+1, li+1): not reboot

f(dj+1, 1): reboot.

Correctness: By our algorithm. When we consider d_i is today. l_i is the days of difference from reboot to d_i. By given the maximum solution must be in either reboot or not reboot. therefore pick the maximum. we loop through n days. case 1. not reboot. we choose the minimum of X_i or S_i (by given), then we plus the value of first day after reboot. if it reboot today, we from date the day, this day we skip to count. then process the s[we the d from (we have found. repeat the step. the return will be optima values.