Run-time analysis (16 pts).

1. (2pts each) In each of the following cases, indicate if $f = O(g)$, $f = \Omega(g)$, $f = \Theta(g)$.

$f(n)$	$g(n)$	O	Ω	Θ
2^{n+10}	2^{2n}	T	F	F.
$\log_5 \sqrt{n}$	$\log_2 n$	T	T	T
$n \log_2 n$	$n^{1.5}$	T	F	F

$f(n) = 2^{10} \cdot 2^{n}$, $g(n) = (2^2)^n = 4^n$

$f(n) = \log_5 n^{1/2} = \frac{1}{2} \log_5 n$

$g(n) = n \cdot n^{1/2}$.

$\log_2 n$ is dominated by $n^{1/2}$.

2. (5 pts each) Please solve the following recurrence relations.

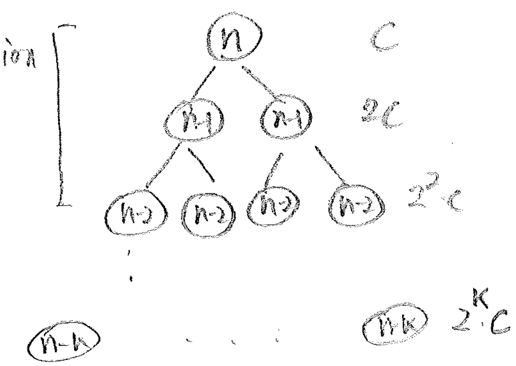
a. $T(n) = 2T(n-1) + c$

b. $T(n) = T(n/2) + cn$

$\boxed{a}$ $T(n) = 2T(n-1) + c = 2(2T(n-2) + c) + c = 2^2 T(n-2) + 2c + c$

$= 2^k T(n-k) + 2^{k-1} c + \cdots + c = c(1+2+2^2+\cdots+2^k) = \Theta(2^k)$.

Recursion Tree



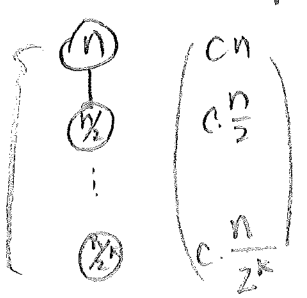
$T(n) = c + 2c + 2^2 c + \cdots + 2^k c = \underline{\Theta(2^k)}$

Note: the end result needs to be in Θ. While $O(\cdot)$ is correct, it is only capturing part of the truth thus considered incomplete.

(b) $T(n) = T(n/2) + c \cdot n = T(n/4) + c\, n/2 + cn = T(n/2^k) + c\, n/2^{k-1} + \cdots + cn$

$= \sum_{i=0}^{\log_2 n} (\frac{1}{2})^i \cdot cn = cn \cdot (\underbrace{1 + \frac{1}{2} + (\frac{1}{2})^2 + \cdots + (\frac{1}{2})^{\log_2 n}}_{\Theta(1)}) = \Theta(n)$

Recursion Tree.



sums up to

$(cn + \frac{1}{2} cn + \frac{1}{2^2} cn + \cdots + \frac{1}{2^k} cn) = \Theta(n)$.

Section	total	score
Runtime	16	

Proof by Induction (8 pts) Prove the following statement by induction.

A tree with n nodes has $n-1$ edges.

Note that here a tree refers to a connected graph that contains no cycles.

Please clearly state the base case, inductive assumption and the inductive step:

Base case: (1pt)

Base case considers the smallest tree i.e. one node. it has no edge.
$n=1$. # edges $= 0 =$ # of nodes -1.

Inductive assumption: (1pt)

Assume that the statement is true for trees with $1, 2, \cdots k$ nodes.

Inductive step: (6pts)

Consider a tree T of $k+1$ nodes.
It will have at least one leave node. i.e. a node with no children.
Remove this node and the edge connecting it from the tree T resulting in T'.

T' must have k nodes.
Inductive Hypothesis applies.
# of edges of $T' = k-1$.

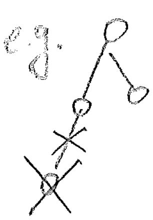
# of edges of $T =$ # of edges of $T' + 1$
$= k-1+1$.

$\Rightarrow$ # of edges of $T =$ # of nodes of $T - 1$

QED

Thought process:
How can we reduce the size of a tree by 1?
Easy, we can take one leave node off. By taking this one node off, we will also need to remove the edge connecting to it. We reduce the # of nodes by 1, and reduce the # of edges by 1. Consider the new tree, it has k nodes, the inductive hypothesis applies.

e.g.



Section	total	score
Proof	8	

Divide and Conquer. (15 pts)
Given an array of n objects $A[1], A[2], ..., A[n]$, you cannot sort them (think of the objects as images, or files), but can compare if two objects are equal in $O(1)$ time. An element of A is called the majority if it occurs more than $\frac{n}{2}$ times (not just that it is the most common) in A. For example $[1,2,2,4]$ does not have a majority element.

a. (10 pts) Design an algorithm that returns either the majority element or return Null if no majority element exists. For full credit, the algorithm must run in $O(n \log n)$ time. You may describe your algorithm in plain English, or in pseudocode. You can assume n is a power of 2. (Hint: if you know the majority elements of the left and right sub-arrays of A, how to figure out the majority element of A?)

Thought process

1. Goal $O(n\log n) \Rightarrow$ the combine step must be cn.
2. General idea: break A into A_L, A_R. Find majority in A_L, 〃 〃 in A_R. In either cases, it can be NULL. Then combine in cn.
3. If an element is majority of A, it has to be majority for one of A_L, A_R.

Key to the design

Majority (A, n)
if $n=1$. return $A[1]$
$A_L = A[1, \dots n/2]$
$A_R = A[n/2, \dots n]$.
$T(n/2) \leftarrow$ $l_m =$ majority $(A_L, n/2)$
$T(n/2) \leftarrow$ $l_R =$ majority $(A_R, n/2)$
if $l_m = l_R$ return l_m
if $l_m \neq$ NULL
$cn \leftarrow$ scan A to see if l_m is majority
if so. return l_m.
if $l_R \neq$ NULL
$cn \leftarrow$ scan A to see if l_R is majority
if so return l_R.
return null.

b. (5 pts) Provide the recurrence relation describing the run time of your algorithm. You don't need to solve it.

$T(n) = 2T(n/2) + cn$

Section	total	score
D&C	15	

3. (15 pts)
You are given a sequence of n binary bits $x_1, \cdots, x_n \in \{0,1\}$. Your output is to be either

- any i such that $x_i = 1$ or
- the value 0 if the input is all 0s.

For example, if your input is $\{0,0,1,1\}$, the output could be either 3 or 4.

The only operation you are allowed to use to access the inputs is a function Group-Test where Group-Test(i,j) returns 1 if any bit in $x_i, x_{i+1}, \cdots, x_j$ has value 1, and returns 0 otherwise. For example, for the given input sequence $\{0,0,1,1\}$, Group-Test$(2,4) = 1$ and Group-Test$(1,2) = 0$.

(Historical Note: In World War I, the army was testing recruits for syphilis which was rare but required a time-consuming but accurate blood test. They realized that they could pool the blood from several recruits at once and save time by eliminating large groups of recruits who didn't have syphilis.)

a) (10 points) Design a divide and conquer algorithm to solve the problem that uses only $O(\log n)$ calls to Group-Test in the worst case. Your algorithm should never access the x_i directly.

b) (5 points) Briey justify your bound on the number of calls to Group-Test.

Thought Process.

1. Goal $O(\log n)$
Need to shrink input by half (or a constant) in constant time like binary search.

2. Note that we only need to find one i s.t. $X_i = 1$, if one exists,
Applying Grouptest once we can either eliminate the tested half (if Grouptest returns 0) ~~of~~ or focus on it (if Grouptest returns 1).

FindOne $(X_1, X_2, \cdots, X_n)$
if $n=1$ return Group-test$(1,1)$
if Grouptest $(X_1, \cdots X_{n/2})$
return FindOne $(X_1, \cdots, X_{n/2})$
else
return FindOne $(X_{n/2}, \cdots X_n) + \frac{n}{2}$

so that the index will be correct

$T(n) = T(n/2) + 1$

$\Rightarrow T(n) = O(\log n)$.

4. Dynamic programming. (20 pts)

Tomorrow is the big dance contest you've been training for your entire life. You've obtained a list of n songs that will be played during the contest, in chronological order. For the k-th song, if you dance to it, you will get exactly $score[k]$ points, but then you will be physically unable to dance for the next $wait[k]$ songs.

Here is an example input, we have:

song	1	2	3	4	5
score	5	10	4	8	5
wait	1	2	1	2	1

That is, if you choose to dance to the first song in the schedule, you will get 5 points but have to skip the next song. If you choose to dance to the second song, you will get 10 points, but have to skip the next two songs. You can choose as many songs as you can subject to the wait-time constraints. The goal is to maximize your total score.

Describe and analyze an efficient algorithm to compute the maximum total score you can achieve given a pair of input arrays $score[1 \ldots n]$ and $wait[1 \ldots n]$.

You must:

(a) (5pts) Define (in words) the subproblems you solve.

(b) (10pts) Give a recurrence relation for the subproblems. Also please provide the *base case*, and state the *final output* of your dynamic programming algorithm.

(c) (5pts) Give the running time of your algorithm. Explain.

Thought process: the input size of this problem is the # of songs we must consider. This suggests a one dimensional array. L(i).

There are two directions we could define L(i):

1. considering songs 1, 2, ⋯ i.

If you try the first one, the difficulty is that if you dance i, there will be a subset in 1,..,i-1 that do not conflict with i, but they might not be contiguous.

2. considering songs i, i+1, ⋯ n.

Because the ~~weigh~~ wait time influence forward choices, it is more natural to consider option 2. L(i) = max score achievable considering songs i, ⋯, n.

i.e. we will sit out songs 1, 2, ⋯ i.

The underlined part are the key information that you must provide in your answer.

What choices do we have for song i?

#1. we sit it out. then we will have subproblem L(i+1).

#2. we dance it. then we will skip. wait(i) songs, and resume at i+wait(i)+1 -th song, which is captured by L(i+wait(i)+1). The value of this option is thus. L(i+wait(i)+1) + score(i).

$$L(i) = \max(L(i+1),\ L(i+wait(i)+1) + score(i))$$

Base case: L(n) = score(n). Final solution: L(1).

Runtime: Each entry of L is constant time computation. overall: O(n)