CS325 — Midterm — Winter 2017

Solutions

Name (First and Last):____

OSU Student ID:____

DO NOT turn to the next page until instructed to do so.

1. There are 7 pages in this exam (including cover page)
2. Please write down your initials on top of EVERY page. You get 1 point for doing that.
3. If using the back of the page, indicate on the front so that it will not be missed.
4. Write your answer concisely and precisely. Keep handwriting clean and easily recognizable.
5. Suggestion: look over the whole exam first and **start with problem you find easiest.**

These formulae may be useful to you:

$$\log_a x = \log_a b \log_b x$$
$$a^{\log_b x} = x^{\log_b a}$$
$$\log_a x \cdot y = \log_a x + \log_a y$$

Problem	Max	score
Initials	1	
Run-time Analysis	16	
Proof by induction	10	
Divide and Conquer	12	
Dynamic Programming	15	
Bonus	6	
Total(Bonus)	54(6)	

Run-time analysis (16 pts). (Graded by Hamed Shahbazi)

1. (0.5 pt each cell) For each case indicate True (T) or False (F) for $f = O(g)$, $f = \Omega(g)$, $f = \Theta(g)$.

$f(n)$	$g(n)$	$f = O(g)$	$f = \Omega(g)$	$f = \Theta(g)$
2^n	$1.5^n n^2$	F	T	F
$2^{n/2}$	2^{n-1}	T	F	F
$1 + 0.5 + 0.5^2 + \cdots + 0.5^n$	$2n$	T	F	F
$n(\log_2 n)^2$	$n^{1.01}$	T	F	F

2. (5 pts each) Please solve the following recurrence relations. Show your steps. If you use master theorem, clearly state which case of the master theorem is used to arrive at your answer.

   a. $T(n) = 3T(n-3) + c$
   $T(n) = \theta(3^{n/3})$. This can be solved using either recursion tree or telescoping.
   For telescoping, 1) showing one step of telescoping is worth 1 pt. 2) Multiple steps with the patterns 3^k, $n - 3k$, will each get one point. 3) Figuring out the depth $k = n/3$ is worth another point. 4) Arriving at the final answer 1 pt. Step 1 can potentially be skipped if later steps were correct.

   For recursion tree: 1) 1 point for the first layer of the recursion tree; 2) general pattern with the correct tree structure with the number of nodes in each level, and the size of each node at each level, one point each; 3) figuring out the depth of the tree $k = n/3$ worth 1 point; 4) putting the final solution together 1 pt.

   b. $T(n) = 4T(n/2) + cn$
   (b.) $T(n) = \theta(n^2)$
   If solved using master theorem, has to clearly indicate how it is used to arrive at the solution.
   If solved using telescoping, 1) showing one step of telescoping is worth 1 pt. 2) Multiple steps with the patterns 4^k, $n/2^k$, will each get one point. 3) Figuring out the depth $k = \log_2 n$ is worth another point. 4) Arriving at the final answer 1 pt. Step 1 can potentially be skipped if later steps were correct.

   For recursion tree: 1) 1 point for the first layer of the recursion tree; 2) general pattern with the correct tree structure with the number of nodes in each level, and the size of each node at each level, one point each; 3) figuring out the depth of the tree $k = \log_2 n$ worth 1 point; 4) putting the final solution together 1 pt.

3. (10 pts) Prove that MergeSort correctly sorts an input array of length n by induction: (Graded by Juan Liu)

```
MergeSort(array $A$)
    if $A$ has one element
        return $A$
    else
        $A_L$ = MergeSort(first half of $A$)
        $A_R$ = MergeSort(second half of $A$)
        return Merge($A_L$, $A_R$)
```

You may assume that Merge(A_L, A_R) outputs a sorted array that contains all the elements in A_L and A_R if A_L and A_R are each sorted. Clearly state the different parts of your inductive proof
Knowing the structure of the proof (need base case, inductive hypothesis and inductive step) is worth 1 pt.

Base case: 2 pts. If A has one element it is already sorted.

Inductive hypothesis: 2 pts Mergesort correctly sort any input array of size $1, ..., k$ *for some* $k \geq 1$.

Inductive step: For input array of size $k+1$ *(1 pt for knowing we need to argue for* $k+1$*), because* $k \geq 1$, $\frac{k+1}{2} \leq k$*, so the inductive assumption applies (2 pts for properly arguing the inductive assumption applies). So* A_L *and* A_R *would be sorted correctly (1 pt to conclude that* A_L *and* A_R *are correctly sorted). Because given two sorted array, Merge will output a sorted array combining* A_L *and* A_R*. The final output will be correct (1 pt for arguing the correctness of the final output based on the assumption that Merge is correct.)*

4. (12 pts) **Definition:**An arithmetic progression is a sequence of numbers in which each number differs from the preceding one by a constant, e.g., {1, 3, 5, 7, 9} and {9, 5, 1, -3}. (Graded by Juneki Hong)
Given an array $A[1, ..., n]$ containing an arithmetic progression with one element missing, use divide and conquer to find the missing element in $O(\log n)$ time.
You can assume that *diff*, the difference between consecutive numbers, is given and input A always has one missing element, which is not the 1st or last one in the progression. Also n is always a power of 2.

For example, for input: 2, 4, 8, 10 with *diff* $= 2$, the output is 6.
For input: 22, 18, 14, 6 with *diff* $= -4$, the output is 10.

You can describe your algorithm in English or pseudo-code. In addition, you also need to analyze the runtime of your algorithm using recurrence relation and show that it runs in $O(\log n)$ time.

```
FindMissing($A[1, ..., n]$, diff)
    if $n < 2$
        return "inappropriate input!"
    if $n = 2$
        return $A[1]$+diff
    $m = \frac{n}{2}$
    if $A[m] \neq A[1] + diff \times (m - 1)$
        return FindMissing($A[1, ..., m]$, diff)
    if $A[m + 1] \neq A[m] + diff$
        return $A[m] + diff$
    return FindMissing($A[m + 1, ..., n]$, diff)
```

Alternatively, we can do:

```
FindMissing($A[1, ..., n]$, diff)
    if $n < 2$
        return "inappropriate input!"
    if $n = 2$
        return $A[1]$+diff
    $m = \lceil \frac{n}{2} \rceil$
    if $A[m] \neq A[1] + diff \times (m - 1)$
        return FindMissing($A[1, ..., m]$, diff)
    return FindMissing($A[m, ..., n]$, diff)
```

Rubrics:

- *Realizing that we need to shrink the input by half each iteration - 1 point*
- *realizing that we need to constant time work - 1 point*
- *Base case: 2 points. The base case for this problem is a bit messy because size 0 and 1 don't really form proper input. If you have only one element, we don't really have a progression. Only when we have 2 elements with known diff it becomes meaningful. Because there is always an element missing, and it can't be the first or the last element of the array. It has to be the middle one. Thus we can return $A[1] + diff$.*
- *The condition for recursion: this is a key part of the design. 3 points*
- *correctly incur the recursive function call on the appropriate input 2 points*
- *correctly argue for the run time 3 points*

5. (15 pts) Dynamic programming. You can choose to solve one of the following problems. (Graded by Souti Chattopadhyay)

Problem A: Maximum Coin collection. You are given a row of n coins with values $v_1, v_2, ..., v_n$. If you collect one coin, you cannot take its neighbors. The goal is to maximize the values of the collected coins. For example, if the row of coins are 5, 10, 1, 5, 10, 25, we should collect the 2nd, 4th, and last coin, leading to a total value of 40.

Problem B: Most efficient Change. Given a fixed denomination of coins, specified by $v_1, v_2, ..., v_k$, and a target B, the goal is to make the change B with the least number of coins. For example, if the denominations are $1, 5, 9, 15$, to make change 18, we should use two 9s.

Your solution should clearly indicate which problem you are solving, and provide the following:

- (3 pts) The definition of your subproblem, in words.
- (6 pts) The procedure for computing the solution for the subproblems, including the base case and the iterative application of the recurrence relations.
- (2 pts) The final solution that your algorithm would return. For problem A, this would be the maximum value you can achieve for the given row of coins. For problem B, this would be the minimum number of coins needed to make the change B.
- (4 pts) An analysis of the runtime of your algorithm.

Problem A.
$L(i)$: the maximum value achievable considering only coins $1, ..., i$

- *Limiting to the first i coins - 2 pts*
- *Explicitly stating $L(i)$ is the maximum or optimal value - 1 pt*

Base case: $L(1) = v_1$ and $L(0) = 0$. - 1 pt
Iterative application of the recurrence relation (assuming base case covers $L(1)$):

```
for $i = 2$ to $n$ - 1 pt
    $L(i) = \max\{L(i-1), v_i + L(i-2)\} - 5pts$
```

Note: for the five points, consider assigning partial credits according to how close they are to figuring out the recurrence relation. They don't have to show the pseudo-code. Describing these in words would be acceptable too. If they don't use the for loop, to get the 1 point, it needs to be clear that their method fills up the array from small i to larger values up to n.

Return the final solution: return $L(n)$ - 2 pts
Runtime: - 4 pts 2pts for recognizing there are n subproblems, 2 pts for recognizing each subproblem can be computed in constant time - $\theta(n)$ or $O(n)$ - either one is ok. If their algorithm is incorrect, they can get up to 3 points depending on the correctness of their runtime analysis.

Problem B.
$L(t)$: the minimum number of coins to make change t

- *Index the array with t, the target - 2 pts*
- *Explicitly stating $L(t)$ is the minimum or optimal number - 1 pt*

Base case: $L(0) = 0$ - 1 pt
Iterative application of the recurrence relation:

```
for $t = 1$ to $B$ - 1 pt
    $L(t) = \min_{i:v_i \leq t} L(t - v_i) + 1$ - 5 pts
//For target $t$, we can use any of the coins with $v_i \leq t$, and optimally solve the resulting subproblem $L(t - v_i)$.
//Considering all possible options, we choose the minimum one.
```

Note: for the five points, consider assigning partial credits according to how close they are to figuring out the recurrence relation. They don't have to show the pseudo-code. Describing these in words would be acceptable too. If they use words, to get the 1 point for the for loop, it needs to be clearly stated that their method fills up the array from small t to larger values up to B.

Return the final solution: return $L(B)$ - 2 pts
Runtime analysis: 4 pts in total. 2pts for recognizing there are B subproblems, 2pts for recognizing solving each subproblem requires in the worst case comparing n options, leading to $\theta(nB)$ or $O(nB)$. If their algorithm is incorrect, they can get up to 3 points depending on the correctness of their runtime analysis.

Bonus questions. (Graded by Hamed Shahbazi)
(3 pts each) Assume you have functions f and g such that $f(n) = O(g(n))$. For each of the following statements, prove it is true or provide a counterexample to show its false.

(a.) $\log_2 f(n) = O(\log_2 g(n))$
(b.) $2^{f(n)} = O(2^{g(n)})$

(a.). *True*
Because $f = O(g)$, we know for some constants c' and n_0', we have $f(n) \leq cg(n)$ for $n \geq n_0$. Take log on both sides of the inequality, we have:

$$\log_2 f(n) \leq \log_2 c + \log_2 g(n) \text{ for all } n \geq n_0$$

We can find an n_1 such that $g(n) \geq c$ for all $n \geq n_1$. Thus we have:

$$\log_2 f(n) \leq 2 \log_2 g(n) \text{ for all } n \geq \max(n_0, n_1) \Rightarrow$$

$$\log_2 f(n) = O(\log_2 g(n))$$

(b.) *False*
Let $f(n) = 2n$ and $g(n) = n$.
We have $f = O(g)$ but $2^{2n} = 4^n$ is not $O(2^n)$

Rubric. *Any nontrivial attempt regardless whether it is correct or not automatically gets 1 pt*
For each subproblem, if answer True/false correctly - 1 pt
To get full credit, correct justification must be give.