CS 341 Midterm

2018-02-27

(Extra space.)

Instructions

- NO CALCULATORS OR OTHER AIDS ARE ALLOWED.
- You should have 34 pages (including the header and extra pages).
- Make sure you fill the information on the header page.
- Solutions will be marked for clarity, conciseness and correctness.
- If you need more space to complete an answer, you may continue on the two blank extra pages at the end.

Useful Facts and Formulas

1. Master Theorem
   Suppose that $a \geq 1$ and $b > 1$, $d \geq 0$. Consider the recurrence

$$T(n) = a\,T\left(\frac{n}{b}\right) + \Theta(n^d)$$

   Then:

$$T(n) \in \begin{cases} \Theta(n^{\log_b(a)}) & \text{if } a > b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^d) & \text{if } a < b^d. \end{cases}$$

2. $a^{\log_b c} = c^{\log_b a}$

(Extra space.)

1. (17 marks) *Short Questions 1* For each question below, give your answer together with a brief explanation. Show computations if it is appropriate to do so.

a) **(5 marks)** Suppose $f(n) = O(n)$, $g(n) = O(n)$, then is $f(n) + g(n) = O(n)$? If so, please provide a proof. Otherwise provide a counter example (i.e., an example $f(n)$ and $g(n)$ that are $O(n)$ but $f(n) + g(n)$ is not $O(n)$).

Solution: By definition of f(n) = O(n): $\exists n_1 \geq 0, c_1 > 0$ s.t. $f(n) \leq c_1 n$ for all $n > n_1$.

By definition of g(n) = O(n): $\exists n_2 \geq 0, c_2 > 0$ s.t. $g(n) \leq c_2 n$ for all $n > n_2$.

Therefore $f(n) + g(n) < (c_1 + c_2)n$ for all $n > \max\{n_1, n_2\}$.

b) **(3 marks)** Order these three functions in increasing order of asymptotic growth rate. You do not need to justify your answer.

$f(n) = \log(n)^{50} + 1.01^n$

$g(n) = \log(n)^{175} + n^{0.550}$

$h(n) = \log(n)^{200} + n^{0.450}$

Solution: $\log(n)^{200} + n^{0.450}$, $\log(n)^{175} + n^{0.550}$, $\log(n)^{50} + 1.01^n$.

(Extra space.)

c) **(3 marks)** Give three examples of divide-and-conquer algorithms. Just list the names of the algorithms or the computational problems they solved. Solution: examples: MergeSort, Karatsuba-Offman integer multiplication, Shamos' Algorithm (2D Closest Pair), Strassen's Algorithm, 2D-Maxima, Counting Inversions DC Algorithm, FFT DC Algorithm

d) **(6 marks)** Consider running the Gale-Shapley algorithm on an input with the following intern and hospital preferences:

Interns	Hospital Ranks				Hospitals	Intern Ranks			
1	A	B	C	D	A	4	1	2	3
2	B	C	D	A	B	1	4	3	2
3	C	D	A	B	C	2	3	4	1
4	D	A	B	C	D	1	2	3	4

- **(3 marks)** When interns propose, what is the matching given by the GS algorithm?
  Solution:
  1-A
  2-B
  3-C
  4-D

- **(3 marks)** When hospitals propose, what is the matching given by the GS algorithm?
  Solution:
  1-B
  2-C
  3-D
  4-A

(Extra space.)

2. (13 marks) *Short Questions 2* For each question below, give your answer together with a brief explanation. Show computations if it is appropriate to do so.

a) **(3 marks)** In the job scheduling problem we had in lectures, where each job had both a weight and a length, and the goal is to minimize the weighted total completion time, what was the optimal greedy strategy? (If your answer says "sort by parameter x", please indicate if you're sorting in increasing or decreasing value.)

Solution: Sort in increasing order of length/weight.

b) **(5 marks)** Consider a sequence $F(n)$ defined by the following recursive formula: $F(n) = F(n-1) + F(n-2)$, with $F(0) = 0$ and $F(1) = 1$. Given n, we want to design an algorithm to compute $F(n)$. Explain why an approach based on blind recursion is a terrible idea, and why dynamic programming yields a faster algorithm. We are asking just for a conceptual answer, not an explicit dynamic programming algorithm or its pseudocode, correctness, and runtime analysis.

Solution: Blind recursion takes exponential times because the same subproblems would be solved over and over again. For example if we expand the recursion tree of a recursive algorithm solving this problem, $F(n)$ would call $F(n-1)$ and $F(n-2)$. $F(n-1)$ would then call $F(n-2)$ and $F(n-3)$, so we would have two recursive calls solving $F(n-2)$. A DP algorithm does not solve the same subproblems over and over again. In this case, there are $O(n)$ distinct subproblems and a DP algorithm would take $O(n)$ time to solve the problem.

(Extra space.)

c) **(5 marks)** Recall the matrix multiplication oder (also referred to as paranthesization) problem. Given matrices $M_1, \ldots, M_n$, where M_i has dimension $d_{i-1} \times d_i$, we need to compute the optimal order of computing $M_1 \times \ldots \times M_n$ that minimizes the total number of numeric multiplications needed. Consider the following pseudocode for the dynamic programming algorithm that computes the minimum number of numeric multiplications needed,

1. procedure DP-Matrix-Ordering$(d_0, d_1, \ldots, d_n)$:
2. Base Cases: $S[i][i] = 0$; $S[i][i+1] = d_{i-1}d_id_{i+1}$
3. for $i = 1, \ldots, n$
4. for $j = i+2, \ldots, n$
5. $x = +\infty$
6. for $k = i, \ldots, j-1$
7. $x = \min(x, S[i][k] + S[k+1][j] + d_{i-1}d_kd_j)$
8. $S[i][j] = x$
9. return $S[1][n]$

What is wrong with this algorithm?

Solution: Wrong order of filling in the DP table. For example, when $i = 1$, some spots say S[2][4] is not filled yet, but line 7 is using it.

(Extra space.)

3. (10 marks) *Recurrences.* Consider the recurrence:

$$T(n) = 4T(n/2) + 2n$$

$$T(1) = 1$$

Prove $T(n) = O(n^2)$ by induction.

Hint: Guess $T(n) = an^2 - bn$ for some a, $b > 0$.

Solution: Choose $a = 3$ and $b = 2$. Claim: $T(n) \leq 3n^2 - 2n$ Base Case: $n = 1$: $T(1) = 1 \leq 3 - 2 = 1$

IH: Suppose for all $k = 1, ..., n-1$ $T(k) \leq 3k^2 - 2k$

Prove: For n: $T(n) \leq 3n^2 - 2n$

$$\begin{aligned} T(n) &\leq 4(3(n/2)^2 - 2(n/2)) + 2n \\ &= 4(3n^2/4 - n) + 2n \\ &= 3n^2 - 4n + 2n \\ &= 3n^2 - 2n \end{aligned}$$

Therefore, $T(n) = O(n^2)$.

(Extra space.)

4. **(20 marks)** *Divide and Conquer.* CHAOS is an international spy organization, which has n agents in separate locations. Each agent has a computer that has an identical copy of an encryption key. An enemy organization attacked CHAOS' system lately, which may have changed the stored keys in some of these computers. Luckily, strictly more than half of the agents' keys are unaffected. CHAOS hired you to find out which agents' keys are unaffected.

   a) **(5 marks)** You have access to a system called `TEST-SAME` that can, through a safe channel, communicate with two agents i and j and tell you if agents i and j have the same key. Given an agent i, design a procedure to determine whether the agent is unaffected with $O(n)$ `TEST-SAME` operations.

   Solution: Test i against all other agents using the `TEST-SAME` operation, and if we get more than half "yes, same" results, then i is unaffected. This is $n - 1$ so $O(n)$ tests.

(Extra space.)

b) **(10 marks)** Design a divide and conquer algorithm, that performs $O(n \log(n))$ `TEST-SAME` operations to find all of the unaffected agents. Give the pseudocode of your algorithm and give brief explanation of its correctness (you'll do runtime analysis in part (c)).

Solution: DC algorithm:
CHAOS(A,n)
Input: Array A containing n elements
if $n = 1$, return the only element as unaffected.
else split A into two arrays A_1 with $a_1 = \lceil n/2 \rceil$ elements and A_2 with $a_2 = \lfloor n/2 \rfloor$ elements.
agent1 = CHAOS(A_1, a_1);
agent2 = CHAOS(A_2, a_2);
if agent1 is unaffected (by TEST-SAME against A, Part (a), return agent1; else return agent2

Proof of correctness: The last sentence guarantees we return the unaffected agent if more than 1/2 agents in A are unaffected, by Part (a) and the following Claim:

Claim. If we divide the n agents into two groups of $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ agents, then at least for one of the two groups, there are more than half of unaffected agents.

Proof of Claim. If each side has less than or equal to 1/2 unaffected agents, then we do not have more than half unaffected agents among n agents, contradiction.

(Extra space.)

c) **(5 marks)** Write down the recurrence for the runtime of your algorithm and analyze its runtime (i.e, # `TEST-SAME` operations). You may use the Master Theorem.

Solution: Time complexity: $T(n) = 2T(n/2) + O(n)$. This is Master Theorem case 2, thus the solution is $T(n) = O(n \log(n))$.

(Extra space.)

5. (20 marks) *Greedy Algorithms.* Consider the following matching problem: Let $A = \{a_1, \ldots, a_n\}$ and $B = \{b_1, ..., b_n\}$ be $2n$ numbers (you can assume for simplicity, they are distinct). Your goal is to match each a_i to one b_j (and vice versa), i.e., form pairs $(a_{i_1}, b_{j_1}), (a_{i_2}, b_{j_2}), \ldots, (a_{i_n}, b_{j_n})$, where $i_1, \ldots, i_n$ and $j_1, \ldots, j_n$ are permutations of $\{1, \ldots, n\}$, such that the following function is *minimized*:

$$\Sigma_{k=1}^{n} |a_{i_k} - b_{j_k}|$$

That is, your goal is to minimize the sum of the absolute values of the differences of pairs. Here are two different greedy strategies to solve this problem:

Strategy 1: Pick the pair (a, b) with the smallest $|a - b|$ value, where $a \in A$ and $b \in B$. Then remove a and b, and repeat.

Strategy 2: Pick the pair (a, b), where a is the smallest number in A and b is the smallest number in B. Then remove a and b, and repeat. Note that we can implement this method in $O(n \log(n))$ time by sorting.

a) **(5 marks)** Give a counterexample showing that one of these strategies is incorrect (i.e., it can sometimes give a non-optimal solution).

Solution: Strategy 1 is incorrect. Consider

$$1, 4$$

$$3, 5$$

. Strategy 1 will pick $(4, 3)$ because it has the shortest absolute value distance and then $(1, 5)$. This will have a sum of $1 + 4 = 5$. However the better matching is $(1, 3), (4, 5)$, which has a sum of $2 + 1 = 3$.

(Extra space.)

b) **(3 marks)** A mathematical fact is if $a < a'$ and $b < b'$, then $|a - b| + |a' - b'| \leq |a - b'| + |a' - b|$. There are 6 possible cases corresponding to the 6 different possible orderings of a, a', b, b'. For example, one ordering is $a < b < b' < a'$. One can prove this fact by considering all 6 cases and showing that the inequality holds in each case. Instead, show only that the inequality holds for the $a < b < b' < a'$ ordering. (You might want to skip this part initially and come back to it after proving part (c)).

Solution:

Case 1: $a < b < b' < a'$. In this case $|a - b| + |a' - b'| \leq |a - b'| + |a' - b|$ because $|a - b| < |a - b'|$ (because $a < b < b'$) and $|a' - b'| < |a' - b|$ (because $a' > b' > b$).

(Extra space.)

c) **(12 marks)** Give a proof that the other strategy is correct (i.e., it always gives an optimal solution).

Hint: Consider an exchange argument that uses the mathematical fact from part (b).

Solution:

Assume w.l.o.g. that a_i and b_j are ordered in increasing order. Note that any algorithm can be seen simply as a permutation of $b_1, ..., b_n$, where S_g is exactly this order as it pairs $(a_1, b_1), (a_2, b_2), ..., (a_n, b_n)$.

Let S^* an arbitrary solution corresponding to the permutation $b_{k_1}, ..., b_{k_n}$, so it pairs $(a_1, b_{k_1}), ..., (a_n, b_{k_n})$. If S^* is not equal to S_g, then there is a consecutive pair $(a_i, b_{k_i} = b_{j'})$ and $(a_{i+1}, b_{k_{i+1}} = b_j)$, where $b_j < b_{j'}$. Therefore S^* incurs the cost of $|a_i - b_{j'}| + |a_{i+1} - b_j|$ from these two pairs. If we swap $b_{j'}$ and b_j, S^* would incur the cost of $|a_i - b_j| + |a_{i+1} - b_{j'}|$, which by the mathematical fact in part (b) can only decrease the cost of S^* (since S^*'s cost due to other pairs stays the same). If we iteratively swap out of order pairs, a) we will eventually stop because there can be at most (n choose 2) possible swaps (corresponding to each (b_i, b_j)); b) S^* will be equal to S_g. Since we imporove S^* along the way, this establishes that S_g is better than S^* (an arbitrary solution), so S_g is the optimal solution.

(Extra space.)

6. (20 marks) *Dynamic Programming.* You are driving an electric car to go from Waterloo to Vancouver along a fix route. There are n stations at fixed locations $0 = d_0 < d_1 < \ldots < d_n$ in the middle that can provide *battery changes*. For simplicity $d_0 = 0$ is Waterloo, and d_n is Vancouver.

At station i (at distance d_i) there is a battery B_i that lasts for some L_i distance and costs C_i dollars. At each station i, you can replace your battery with the battery B_i and pay C_i dollars or continue driving with the remaining of the current battery you have. Using a dynamic programming algorithm, compute the minimum cost for completing this trip. At the end of the trip, any extra battery power that is left in the battery is worth nothing. You can also assume that there is a solution to the problem, i.e., after picking the first battery at d_0, there is a sequence of battery swaps that can make the car reach Vancouver).

a) **(10 marks)** Let $D[i]$ be the minimum cost to reach the station at d_i. Give a recurrence relation to compute $D[i]$. Briefly justify the correctness.

Solution: $D[i] = \min_{j<i \text{ s.t.} d_j + L_j \geq d_i}(D[j] + C_j)$

To reach station i, I must first reach a station j before i and change battery. The battery power must support me to drive distance $d_i - d_j$. The cost is $D[j] + C_j$. The optimal cost $D[i]$ is the minimum cost among all such choices of j.

(Extra space.)

b) **(10 marks)** Based on (a), provide the pseudocode for the dynamic programming algorithm for this problem. Analyze the time complexity.

Solution:
D[1] = 0
for i from 1 to n
    $D[i] = \min_{j<i \text{ s.t.} d_j+L_j \geq d_i} D[j] + C_j$
Output $D[n]$

Computation of each $D[i]$ takes $O(n)$ time. There are n different i. Therefore, the time complexity is $O(n^2)$.

(Extra space.)

(Extra space.)

(Extra space.)

(Extra space.)

(Extra space.)