Joyce Wang, Matt Demusz, Frank Porco

CS 344: DESIGN AND ANALYSIS OF COMPUTER ALGORITHMS SPRING 2014

Homework 2

Divide-and-Conquer Algorithms, Sorting Algorithms, Greedy Algorithms

Deadline: March 14, 11:59pm.
Available points: 110. Perfect score: 100.

Homework Instructions:

Teams: Homeworks should be completed by teams of students - three at most. No additional credit will be given for students that complete a homework individually. Please inform Athanasios Krontiris **if your team has changed from the previous assignment.** (email: ak979 AT cs.rutgers.edu).

Submission Rules: Submit your solutions electronically as a PDF document through `sakai.rutgers.edu`. Do not submit Word documents, raw text, or hardcopies etc. Make sure to generate and submit a PDF instead. Each team of students should submit only a single copy of their solutions and indicate all team members on their submission. Failure to follow these rules will result in lower grade in the assignment.

Late Submissions: No late submission is allowed. If you don't submit a homework on time, you get 0 points for that homework.

Extra Credit for LaTeX: You will receive 10% extra credit points if you submit your answers as a typeset PDF (using LaTeX, in which case you should also submit electronically your source code). Resources on how to use LaTeX are available on the course's website. There will be a 5% bonus for electronically prepared answers (e.g., on MS Word, etc.) that are not typeset.

25% Rule: For any homework problem (same will hold for exam questions), you can either attempt to answer the question, in which case you will receive between 0 and 100% credit for that question (i.e., you can get partial credit), or you can write "I don't know", in which case you receive 25% credit for that question. Leaving the question blank is the same as writing "I don't know." You can and will get less than 25% credit for a question that you answer erroneously.

Handwritten Reports: If you want to submit a handwritten report, scan it and submit a PDF via Sakai. We will not accept hardcopies. If you choose to submit handwritten answers and we are not able to read them, you will not be awarded any points for the part of the solution that is unreadable.

Precision: Try to be precise. Have in mind that you are trying to convince a very skeptical reader (and computer scientists are the worst kind...) that your answers are correct.

Collusion, Plagiarism, etc.: Each team of students must prepare its solutions independently from other teams, i.e., without using common notes or worksheets with other students or trying to solve problems in collaboration with other teams. You must indicate any external sources you have used in the preparation of your solution. Do not plagiarize online sources and in general make sure you do not violate any of the academic standards of the course, the department or of the university (the standards are available through the course's website). Failure to follow these rules may result in failure in the course.

Part A (20 points)

Problem 1: The more general version of the Master Theorem is the following. Given a recurrence of the form:

$$T(n) = aT(\frac{n}{b}) + f(n)$$

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function, there are 3 cases:

1. If $f(n) = O(n^{log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{log_b a})$.
2. If $f(n) = \Theta(n^{log_b a} log^k n)$ with $k \geq 0$, then $T(n) = \Theta(n^{log_b a} log^{k+1} n)$. In most cases, $k = 0$.
3. If $f(n) = \Omega(n^{log_b a + \epsilon})$ with $\epsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$. The regularity condition specifies that $af(\frac{n}{b}) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

Give asymptotic bounds for the following recurrences. Assume $T(n)$ is constant for $n = 1$. Make your bounds as tight as possible, and justify your answers.

A. $T(n) = 2T(\frac{n}{4}) + n^{0.51}$
B. $T(n) = 16T(\frac{n}{4}) + n!$
C. $T(n) = \sqrt{2}T(\frac{n}{2}) + logn$
D. $T(n) = T(n-1) + lgn$
E. $T(n) = 5T(n/5) + \frac{n}{lgn}$

Part B (25 points)

Problem 2: You are in the HR department of a technology firm, and here is a job for you. There are n different projects, and n different programmers.

Every project has its unique payoff when completed and level of difficulty (which are uniform, regardless which programmer will work on the project). Every programmer has a unique skill set as well as expectations for compensation (which are uniform, regardless the project the programmer will work on). You cannot directly collect information that allows you to compare the payoff or difficulty level of two projects, or the capability or expectations for compensation of two programmers.

Instead, you can arrange a meeting between each project manager and programmer. In each meeting, the project manager will give the programmer an interview to see whether the programmer can do the project; the programmer can ask the project manager about the compensation to see whether it meets her expectations. After the meeting, you can get a result based on the feedback of the project manager and the programmer. The result can be:

1. The programmer can't do the project.
2. The programmer can do the project, but the compensation of the project doesn't meet her expectation.
3. The programmer can do the project, and the compensation for the project matches her expectations. At this time, we say the project and the programmer *match* with each other.

Assume that the projects and programmers match one to one. Your goal is to match each programmer to a project.

A. Show that any algorithm for this problem must need $\Omega(n \log n)$ meetings in the worst case.

B. Design a randomized algorithm for this problem that runs in expected time $O(n \log n)$.

Part C (40 points)

Problem 3: A nation-wide programming contest is held at k universities in North America. The i^{th} university has m_i participants. The total number of participants is n, i.e., $n = \sum_{i=1}^{k} m_i$. In the contest, participants have to write programs to solve 6 problems. Each problem contains 10 test cases, each test case is worth 10 points. Participants aim to maximize their collected points.

After the contest, each university sorts the scores of participants belonging to it and submits the grades to the organizer. Then the organizer has to collect the sorted scores of participants and provide a final sorted list for all participants.

1. For each university, how do they sort the scores of participants belonging to it? Please briefly describe a comparative sorting algorithm that is appropriate for this purpose and a non-comparative sorting algorithm that works in this setup.

   Comparitive Sort: For the comparitive sort, the universities could use merge sort. For this sort, they would take an array of unsorted scores, and split the array in half, then split those halves in half, and so on. This would be done recursivley until they have all singleton arrays. Now that they have all singleton arrays, the actual comparitive work begins. They would work their way back up the tree that was created from all of the splits. Starting at the bottum, they would compare and sort the scores that were split last. This would be done for every pair at the bottum of the tree. Then the same would be done on the next level of the tree, comparing and sorting the small sorted arrays that were just created. After each level of the tree is completed, they would have fewer and larger sorted arrays, until they get back up to the top of the tree with one final sorted array of all of that universities scores.

   Non-Comparitive Sort: For the non-compartive sort, the universities could use counting sort. Counting sort could work for this application because the range that the scores could be in is known. They would start by creating an array of integers with a length matching the range of scores, with every index initilizied to 0. Then they would run through the array of scores, and for each score, go to the corresponding location in the created array and increment it. After that step is completed, the values in the new array will be altered. Instead of each index holding the number of scores equal to that index, they will each hold the number of scores that are equal to or less than that index. This would be done by starting at the begining of the new array. The first index will remain the same. The second index will be set to its own value added to the value at the first index. The third index will be set to its own value added to the new value of the second index. This will continue for every index in the new array. The last step will begin with creating another new array of integers with the length matching the number of total scores from that university.

2. How does the organizer sort the scores of participants given k files, where each file includes the sorted scores of participants from a specific university? Please describe an algorithm with a $O(n \log k)$ running time and justify its time complexity.

3. Suppose the organizer want to figure out the participants of ranking r. Given k sorted files, how does the organizer find the rth largest scores without sorting the scores of participants? Please describe an algorithm in $O(k(\sum_{i=1}^{k} \log m_i))$ time and justify its time complexity.

   Could you do this in $O(k + \log k \sum_{i=1}^{n} \log m_i)$ time?

   [Hint: If r is larger than $\frac{n}{2}$, the elements that have at least $\frac{n}{2}$ elements larger than them should not be possible answer. The problem is how to identify these elements.]

Problem 4: You have a collection of n New York Times crossword puzzles from 01/01/1943 until 12/31/2012 stored in a database. The only operations that you can perform to the database are the following:

- crossword_puzzle $x \leftarrow$ getPuzzle(int index); where the index is between 1 and n; the puzzles are *not* sorted in the database in terms of the date they appeared.
- getDay(crossword_puzzle x); which returns a number between 1 to 31.
- getMonth(crossword_puzzle x); which returns a number between 1 to 12.
- getYear(crossword_puzzle x); which returns a number between 1943 to 2012.

All of the above queries can be performed in constant time. You have found out that the number of puzzles is less than the number of days in the above period (from 01/01/1943 until 12/31/2012) by one, i.e., one crossword puzzle was not included in the database. We need to identify the date of the missing crossword puzzle.

Design a linear-time algorithm that minimizes the amount of space that it is using to find the missing date. Ignore the effect of leap years.

1. Start by creating an array of integers (all inititilized to 0) with a length of the number of years (1943 - 2012).
2. Go through the entries in the database using getPuzzle() starting at zero, and perform a getYear() on all of them. For every puzzle, go to the array index corresponding to that puzzles year and increase it by 1.
3. Traverse the array and find the year (array index) that doesn't have a value of 365. This is the year of the missing puzzle.
4. Create another array of length 12 (number of months) and inititialize each index to 0.
5. Go through the entries in the database again and do a getYear on them. If the year matches the year of the missing puzzle, do a getMonth() on it. Then go to the month array index corresponding to the month and increment the value at that index.
6. After that, go through the month array and see which month has one less day than it should have. This is the month of the missing puzzle.
7. Create another array with a length matching the number of days in the missing month and inititialize each index to 0.
8. Go through the entries in the database again and do a getYear() on them. If the year matches the year of the missing puzzle, do a getMonth() on it. If the month matches the month of the missing puzzle, do a getDay() on it. Then go to that index of the array of days and increment the value.
9. Then go through the day array and find which index has a value of 0. This is the day of the missing puzzle.
10. Now you have the date of the missing puzzle.

Part D (25 points)

Problem 5: You are running a promotional event for a company during which the plan is to distribute n gifts to the participants. Consider that each gift i is worth an integer number of dollars a_i. There are m people participating in the event, where $m < n$. The j-th person is satisfied if he receives gifts that are worth at least s_j dollars each. The task is to satisfy as many people as possible given that you have a knowledge of the gift amounts a_i and the satisfaction requirements of each person s_j. Give an approximation algorithm for assigning rewards to people with a running time of $O(m \log m + n)$. What is the approximation ratio of your algorithm and why?

Let n be the number of gifts and m be the number of people participating in the event, where $m < n$. To satisfy as many people as possible given the gift amounts a_i and the satisfaction requirements of each person s_j, we will sort gifts in terms of their prices from lowest price to highest price, using radix sort. Because there are n gifts, and radix sort is linear, we have n as our running time now. To continue, we will build a min heap with the people's satisfaction requirements, and building a min heap is linear time, so we have a running time of m in addition to the n from sorting the gifts. To proceed, we will take the minimum satisfaction requirement from the heap, which requires a worst case running time of $log(m)$ to traverse the heap. We will take the minimum satisfaction requirement and compare it to first value – smallest value – in sorted gifts. If that gift value is greater than or equal to that person's satisfaction requirement, then we remove that person from the heap, and we remove the gift from the list. If that gift value is not greater than or equal to that person's satisfaction requirement, then we continue traversing the sorted gifts, and whenever we hit a gift that satisfies the person, then we remove the person from the heap and the gift from the list. Continue removing the minimum value from the heap until each person gets a gift or until no gift satisfies that person. Therefore, this algorithm will satisfy as many people as possible with the running time of $O(mlogm + n)$.