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

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. $T(n) = 2T(\frac{n}{4}) + n^{0.51}$

ANSWER:
From $T(n) = 2T(\frac{n}{4}) + n^{0.51}$, we have $a = 2$, $b = 4$, and $f(n) = n^{0.51}$. Therefore, $n^{log_b a} = n^{log_4 2} = n^{0.5}$. We compare $n^{0.51}$ to $n^{0.5}$, and we know that $n^{0.51} > n^{0.5}$, so it satisfies case 3 because $f(n) = n^{0.51}$, which is bounded below by $n^{0.5}$. Now we have to check the regularity condition, so $2(\frac{n}{4})^{0.51} \leq c(n^{0.51}) = 2(\frac{n^{0.51}}{4^{0.51} n^{0.51}}) = \frac{2}{4^{0.51}} = 0.986 \leq c$. Therefore, the regularity condition is satisfied, so $T(n) = \Theta(f(n)) = \Theta(n^{0.51})$.

B. $T(n) = 16T(\frac{n}{4}) + n!$

ANSWER:
From $T(n) = 16T(\frac{n}{4}) + n!$, we have $a = 16$, $b = 4$, and $f(n) = n!$. Since a running time of n^n is worse than $n!$, we can let $f(n) = n^n$ for now. Therefore, $n^{log_b a} = n^{log_{16} 4} = n^2$, so it satisfies case 3 because $f(n) = n^n$, which is bounded below by n^2. Now we have to check the regularity condition, so $16(\frac{n}{4})^n \leq c(n^n) = 16(\frac{n^n}{4^n}) \leq c(n^n) = 16(\frac{n^n}{4^n n^n}) = (\frac{16}{4^n}) \leq c$. For all sufficiently large n, $c < 1$. Therefore, the regularity condition is satisfied, so $T(n) = \Theta(n!)$.

C. $T(n) = \sqrt{2}T(\frac{n}{2}) + logn$

ANSWER:
From $T(n) = \sqrt{2}T(\frac{n}{2}) + logn$, we have $a = \sqrt{2}$, $b = 2$, and $f(n) = logn$. Therefore, $n^{log_b a} = n^{log_2 \sqrt{2}} = n^{0.5}$, so it satisfies case 1 because $f(n) = logn$, which is bounded above by $n^{0.5}$. So $T(n) = \Theta(n^{log_b a}) = \Theta(n^{log_2 \sqrt{2}}) = \Theta(n^{0.5})$.

D. $T(n) = T(n-1) + lgn$

ANSWER:
This problem does not fit the general version of the Master Theorem, so we can't apply the Master Theorem for this problem. From $T(n) = T(n-1) + lgn$, we can conclude that $T(n-1) = T(n-2) + lg(n-1)$, $T(n-2) = T(n-3) + lg(n-2)$, $T(n-3) = T(n-4) + lg(n-3)$, etc. Therefore, from all those recursive calls, we have the summation $T(n) = \sum_{i=1}^{n} f(n) = \sum_{i=1}^{n} f(n)$. $f(n) = lgn$, so we have $\sum_{i=1}^{n} lgn = lg(n!)$. We can apply Stirling's formula to $lg(n!)$, and since we know that Stirling's formula is $lg(n!) = nlg(n) - n + O(lg(n))$ and has a worst case running time of $O(nlgn)$, we can conclude that $T(n) = O(nlgn)$.

E. $T(n) = 5T(n/5) + \frac{n}{lgn}$

ANSWER:
From $T(n) = 5T(n/5) + \frac{n}{lgn}$, we have $a = 5$, $b = 5$, and $f(n) = \frac{n}{lgn}$. Therefore, $n^{log_b a} = n^{log_5 5} = n$, so it satisfies case 1 because $f(n) = \frac{n}{lgn}$, which grows smaller and smaller for sufficiently large n. It is bounded above by n, so $T(n) = \Theta(n^{log_b a}) = \Theta(n^{log_5 5}) = \Theta(n)$.

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.

ANSWER:
$n! \leq l \leq 2^n$, $n! \leq 2^n$, $\log_2 n! \leq \log_2 2^n$, $\log_2 n! \leq ?(nlog_2 2)$, $\Theta(n \log_2 n) \leq n$, $\Theta(n \log_2 n) \leq n$, $\Omega(n \log_2 n) = n$

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

ANSWER:
This algorithm is very similar to quick sort. Pick a random project, and ask every programmer if they can do the job. Sort the programmers into three lists:

1. Ones that can't do the project
2. Ones for whom the compensation doesn't meet expectation
3. The right programmer for the project

Then take the right programmer from above and ask about each of the non-matching projects. Sort the projects into two lists:

1. Ones that the programmer can do (but don't offer the right compensation)
2. Ones that the programmer can't do

These two steps are done to give some organization to the data (because the problem specifies that you can't directly compare/sort each group).

Then, take a random programmer from 2 (compensation not enough) and repeat the above process with the jobs from the second 2 (the originally matched programmer can't do, i.e. skill isn't good enough). You don't have to compare the group 2 programmers with the group 1 projects because you already know that compensation isn't going to match. You repeat this until all group 2 programmers and projects are matched.

At this point, in the average case, about half of your programmers and projects are matched. Then, you repeat the above process with the programmers and projects from each group 1.

This algorithm is basically doing quicksort (divide and conquer) twice. Quicksort has an expected runtime of $O(n \log n)$. $\log n$ comes from breaking the lists into two each time. n comes from having to do the process for each element in the lists. Because constants are dropped, the above algorithm also has an expected runtime of $O(n \log n)$.

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.

   ANSWER:

   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 bottom, 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 (we'll call array B) 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 (we'll call array C) with the length matching the number of total scores from that university. This new array will be the sorted output. Then they would start at the back of the unsorted array. They would take that score at the last index of the unsorted array and go to that index in array B. Then they would go to the index of array C corresponding with this value. Then the score would be placed at this index, and then value that was pulled from the index in array B will be decremented. This completes placing one number in sorted order in array C. This process will then be repeated for the score at the second to last index in the unsorted array, and so on.

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.

   ANSWER:
   Essentially, we have k sorted lists that we have to merge together into one sorted list. We put a pointer at the first element from each list, and make a min heap using all those values, but we have to keep track of which list that element came from. To form our one sorted list, we remove the minimum element from the heap and add it to the one sorted list, and we have to make sure the move our pointer from that element's original list over to the next element in that same list. Once the pointer is moved to the next element, we take that element and add it to the heap. We repeat this process until all the elements are added to the final list.

   We have to iterate through all of the scores n to add them to the heap, which requires n as the running time. Then, for each score, we have to add it to the heap, which has height $logk$. Therefore, we have a total running time of $O(n \log k)$.

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 $O(k + \log k \sum_{i=1}^{k} \log m_i)$ 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.]

   ANSWER:
   We begin with k sorted files, and we will be applying the median of medians algorithm. To find the rth largest scores, we only consider the r largest scores in each list. We take the first list

and find the median m using $\frac{r}{2}$ if r is even. If r is odd, then $m = \frac{r}{2} + 1$. m is our pivot. Therefore, in each list, we take ALL the values in all the lists higher than m, which we can put into an array h. Our recursive function findMedian is defined as: If r lies between the size of h and the size of h+m, then we stop; our median is equal to the rth largest score. If r is greater than or equal to the size of h, then we call findMedian on the values less than m. If r is less than the size of $h + m$, then we call findMedian on the values greater than m. Once we get m to equal r, then we know that it is the rth largest score. This algorithm is $O(k(\sum_{i=1}^{k} \log m_i))$ since we have at most k lists, and in the worse case, for each list, we call findMedian, which is $logm$.

As for whether or not we can do this in $O(k + \log k \sum_{i=1}^{k} \log m_i)$ time, we don't know!

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.

ANSWER:

1. Start by summing up all the years from 1943 to 2012, and save that value into a variable, totalYearSum.
2. Go through the entries in the database using getPuzzle() starting at zero, and perform a getYear() on all of them. For every puzzle, take totalYearSum and subtract from it the value you get from getYear(), and save the new value into totalYearSum. By the time all the entries are iterated through and the subtraction is done, you'll end up with totalYearSum equalling the year that you're missing.
3. Sum up the months from 1 (January) to 12 (December), and save that value into a variable, totalMonthSum.
4. 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. For every puzzle that you do a getMonth() on, take totalMonthSum and subtract from it the value you get from getMonth(), and save the new value into totalMonthSum. By the time all the entries are iterated through and the subtraction is done, you'll end up with totalMonthSum equalling the month that you're missing.
5. Since you know what month you're missing, you know how many days are in that month (ignoring the fact that it could be a leap year). Sum up from 1 to the number of days there are in the month, and save that value into a variable, totalDaySum.

6. 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. For every puzzle that you do a getDay() on, take totalDaySum and subtract form it the value you get from getDay(), and save the new value into totalDaySum. By the time all the entries are iterated through and the subtraction is done, you'll end up with the totalDaySum equalling the day that you're missing.

7. Now you have the date of the missing puzzle since we have the variables totalYearSum, totalMonthSum, and totalDaySum, and this was done with constant space and linear time since going through the database entries each time is $O(n)$.

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?

ANSWER: 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. We assume that the values of the gifts are in reasonable range and not too expensive, so radix sort is linear in this case. 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 since the heap has a height of $log(m)$. 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, adding those prices to the total sum until you get a sum greater than or equal to the person's satisfaction price. Then we remove the person from the heap and the gift(s) from the list. Continue removing the minimum value from the heap until each person gets a gift or until you run out of gifts. Therefore, this greedy algorithm will satisfy as many people as possible with the running time of $O(mlogm + n)$.

We don't know what the approximation ratio of the algorithm is!