Homework 2

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

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

Hua Yang
Alexio Mota
Erik Kamp

Homework Instructions:

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

- $af(n/b) < cf(n) a = 2 b = 4 2 f(n/4) < cf(n) where c < 1$
- $T(n) = \Omega(f(n)) T(n) = 2T(n/4) + n 0.51 = n 0.51$
- $\Omega(n log b(a + e)) \Omega(n log 4(2)) + e) = f(n)$
- $\Omega(n log b(a + e)) \Omega(n 0.5 + e) = n 0.51$ which proves that $f(n) = \Omega(n log b(a) + e)$ where $e < .01$
- Therefore we can apply case 3 in this situation and get that $T(n) = \Theta(n 0.51)$

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

Answer

- Apply Case 3
- $T(n) = \Omega(f(n)) T(n) = 16T(n/4) + n! = n!$
- $f(n) = \Omega(n log b(a) + e) n! = (n log 4(16) + 3) n! = \Omega(n 2 + e)$

- By condition: $16f(n/4) <= cf(n) where c < 1$.
- We apply case 3 and get that $T(n) = \Theta(n!)$

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

Answer

- Case 1 $\Theta(\sqrt{n})$
- $f(n) = O(n^{logb(a)-e})$ where $a = \sqrt{2}$ and $b = 2$ and $f(n) = log(n)$
- $O(n^{log2(\sqrt{2})+e}) = O(n^{12-e}) = F(log(n))$
- $O(n^{1/2-e}) = f(log(n))$ we can say from the book that any polynomial dominates any logarithm. (Common sense rules page 16 rule 4)
- We apply case 1 and get that $T(n) = \Theta(n)$

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

Answer

- Apply case
- $a = 1$,b =?, $log_b(a) = 0$ since $a = 1$, (aka: $b^0 = 1$)
- $f(n) = log(n) = \Theta(n^{logb(a)} log^k(n))$
- Where k is 1
- $\Theta(n^0 log^1(n)) = \Theta(log(n))$
- Therefore we can say that $f(n) = (n^{logb(a)} log^k(n))$ because $log(n) = \Theta(log(n))$
- Therefore we apply case 2 to get $T(n) = \Theta(n^{logb(a)} log^{k+1}(n)) = \Theta(log^{k+1}(n)) = \Theta(log^2(n))$

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

Answer

- $a = 5$, $b = 5$ and $f(n) = n/(log(n))$
- $n^{log5(5)} n^1 n^{1(+-)e}$
- $n/log(n) = nlog^k(n)$ where most time $k = 0$ then âĘŠ $n/log(n) = \Theta(n)$
- $n/log(n) = O(n^{1-e})$
- Therefore we can apply case 1 since $n/log(n)$ is upper bounded by n^{1-e}. From this case we can get that $T(n) = \Theta(n^{logb(a)}) = \Theta(n^1)$

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

- with x number of project managers and x number of programmers, there are x! permutations
- use binary search as the algorithm, there are at most $2n$ leaves where n is the height of the tree in worst case
- $2^n >= x!$
- $n >= log_2 x!$, by taking the log_2 of both sides
- since $log_2 x! = (x log_2 x)$
- therefore, $n = \Omega(n log_2 n)$

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

Answer

- apply Master Theorem
- pick a project manager and partition the developers with respect to this project manager, in order to find a match
- pick a developer and partition the project managers with respect to this developer, in order to find a match
- for each partition, split the selected group of developers (or project managers depending on the case) into two, stop once a match is found
- $T(n) = T(x) + T(n-x) + O(n)$, by spliting into 2 sub-problems
- average run-time is $O(nlogn)$

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

   - Comparative : Merge sort should be used for the comparative method of sorting because it will only take O(nlog(n)) time to sort a list of n scores. Whereas quicksort will have a worst case runtime of O(n2). Additionally Merge sort works best with sequentially accessed elements like a stack of scores and will preserve the order of the scores.
   - Non-Comparative : Radix sort should be used for the non-comparative method. Radix sort works for this situation because scores can be separated by their integer or digit number. In this sort the scores will be collected and then separated by their value into k groups or buckets. The groups or buckets will be based on the least or most significant digit of the score. These groups are based on the value of the score so if two participants have the same score they will be in the same group. This non-comparative sort will be faster than the comparative sort only taking O(nk) time where k is the number of groups or buckets and n is the number of digits.

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

   - The algorithm to sort the scores of participants given k files would involve a min-heap. You start off with your input lists which can be viewed as a 2D array of scores.
   - We initialize the heap with stores index j for a list and the index i where i,j correspond to a score in list j. The heap has functions insert(i,j) and deleteMin() which returns a i,j pair. We start off by populating the heap with the pair 0,j so the heap will have k entries where k is the number of lists. This runs $O(k)$ times, inserting into the heap would take $O(logk)$, so total runtime is $O(k * logk)$.
   - Now we will proceed to delete the min score from the heap and insert into a result list, result. The min score will be inserted at index c where c starts at 0 and is incremented after each insertion. This will be a loop that runs $O(n)$ times where each iteration a min pair i,j is deleted from the heap and inserted into the result list, result. Deletion from the heap would take $O(logk)$. Insertion and deletion in each iteration would take $O(logk + logk)$
   - Once we delete a min pair from the heap, we will insert i++,j from list j corresponding to the delete index i score into the heap. The reason we do this is because since we have deleted the min score represented by i from the heap, we assume that the next smallest value can be the score in j at index i++ since it is sorted. Pair i++,j will only be inserted if it is not the end of the list.
   - The run time would be $O(n(logk + logk)) = O(nlogk)$.

3. Suppose the organizer want to figure out the participants of ranking r. Given k sorted files, how does the organizer find the r^{th} largest score without sorting the scores of participants? (e.g. if r = 5 you need to compute the 5^{th} largest score, not the top 5 scores). 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 $O(\log k(\sum_{i=1}^{k} \log m_i + k))$ time?

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

Answer

- Assuming that all the lists are sorted in the same order in this case least to greatest the organizer can start off by looking at the last element or score in each sorted list. The organizer must look at the last score in each list erasing a previously greater score when they come to an even greater score (like if the greatest was 50 and comes across a 70). Once the organizer reaches the end of the list they note down that rounds largest and move the end of the list where they found the greatest down / forward one so that they do not repeat that number. The organizer repeats this process until they've reached their desired rth largest score.
- Proceedure :

```
for r != 0 ; r- -
{
    //find greatest from list of sorted
    for i = 0; i < numberOfLists; i++
    {
        //Get the element in the last pos of the list
        if ((listsOfLists[i]).get(listOfLists[i].lastListPointer) > currentMax)
        {
            currentMax =((listsOfLists[i]).get(listOfLists[i].lastListPointer)
        }
    }
    increaseEndOfListContaining(currentMax)
}
return currentMax
```

- This algorithm will use k binary searches looking for the rth largest from k sorted lists. Because binary search takes $O(log(n))$ time the total time should take k time to look through the k lists last element and log(n) time to search through the k elements each loop finding the largest. Therefore the total runtime will wind up taking $O(k * SumOf(log(m_i), 0 <= i <= k)$ time where there are k lists and n elements in total.

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

- Let n be the number of dates between 01/01/1943 and 12/31/2012. We use three variables (TotalDaySum, TotalMonthSum, TotalYearSum) to represent the sum of the values from 01/01/1943 and 12/31/2012. For example, TotalMonthSum represents the sum of months 01-12 for each year. The sum for one year would be 1+2+3+4+...+12 = 78. To the TotalMonthSum would be 78*(100-43+12)=78(69)=5382. Now if 1 date is missing, we can get the MonthSum for the available dates and perform TotalMonthSum-MonthSum to get the month of the missing date. For example, if the missing data is 09/12/2001 the MonthSum would be 5373. By performing TotalMonthSum-MonthSum you get 5382-5373=9 which is the month of the missing date. Repeat this process for the daysum and the year sum to get the full date.
- Pseudocode:

```
TotalDaySum, TotalMonthSum, TotalYearSum

// Go through dates 01/01/1943 and 12/31/2012
getTotal(dates):
    for(i -> dates.length) //takes O(n)
        day, month, year at dates[i]
        TotalDaySum <- TotalDaySum + day
        TotalMonthSum <- TotalMonthSum + month
        TotalYearSum <- TotalYearSum + year

DaySum, MonthSum, YearSum
for( i -> n): // takes O(n)
    crosswordPuzzle x <- getPuzzle(i)
    day <- getDay(x)
    month <- getMonth(x)
    year <- getYear(x)
    DaySum = DaySum + day

    MonthSum <- MonthSum + month
    YearSum <- YearSum + year
endfor

missingDay <- TotalDaySum - DaySum
missingMonth <- TotalMonthSum - MonthSum
missingYear <- TotalYearSum - YearSum

MissingDate = missingDay+âĂİ/âĂİ + missingMonth + âĂIJ/âĂİ + missingYear
```

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

- Since we have m people attending the promotional event and n gifts we can do the following:
  - Sort the m people by their satisfaction requirements, which if using merge sort will take $O(mlog(m))$ time.
  - Secondly we radix sort all the gifts by their gift values. Radix sort will take $O(nk)$ time where k is the number of groups and n is the number of presents. However we can state that n > k through the assumption that the gifts have similar values and are not too sparse. Therefore we can drop this runtime down to $O(n)$.
  - Our approximation value comes in when selecting the correct gift for the correct participant. When selecting the gift we will use the greedy algorithm selecting a gift that is closest to the participants satisfaction value. Once the gift is selected for the participant subtract the gifts value and repeat on the same participant until the satisfaction value for the participant is less than or equal to 0.
- Total runtime : sorting participants : $O(mlog(m))$ + radix Gifts $O(n) = O(mlog(m) + n)$