Assignment 2

Paul Jones and Matthew Klein
Professor Professor Kostas Bekris
Design and Analysis of Computer Algorithms (01.198.344)

March 16, 2014

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:

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

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

- $a = 2$, $b = 4$, where $a \geq 1$ and $b > 1$
- $f(n) = \Omega(n^c)$ where $c = 0.51$
- $\log_b a = \log_4 2 = \frac{1}{2}$
- $c > \log_b a$, that is $0.51 > \frac{1}{2}$
- $\frac{n^{0.51}}{2} \leq cn^{0.51}$ for any constant $c < 1$ and $c > 0.5$
- $T(n) = \Theta(f(n)) = \Theta(n^{0.51})$

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

I don't know.

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

I don't know.

D. $T(n) = T(n-1) + \log n$

I don't know.

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

I don't know.

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.**

There are a total of n projects and n programmers. Since each programmer can only work on one job (regardless of outcome), and there are a total of 3 outcomes. This gives us part of Master's Theorem: $T(n) = a$ where $a = 3$. We need to find the size of each project, which we can guess to be $b = 3$ because we know the answer already ($n \log n$). In fact, this answer makes sense because based on the responses of the programmer's we can find what ones they can do and what ones they can't. We can assume the cost done outside of the calls is instant (meaning talking to the manager's, we aren't concerned with this). Per Master's Theorem: $T(n) = 3(\frac{n}{3}) + 1 \rightarrow \log_3^3 = 1$ which is true, so our answer is $n^1 \log n$.

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

Our randomized algorithm can just be merge sort, we group the programmers in set's, where one set can't do the problem, one set can but wants more pay, and the last set can. Based on their responses, we have our groups to "merge". This algorithm is random because worst, average, and best are $O(n \log n)$.

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.**

   We can use merge sort or quick sort for our comparison based sorting technique. If we want to sort without it, we can use either count or radix because we know the range of the scores.

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.**

   Because each file is already sorted, we need to use a stable sort. Because it is $O(n \log k)$ we must use something such as merge sort or quick sort. Both of these have running time of $O(n \log k)$ in the average cause because each time we split. In this case, there are n participants and k files of an arbitrary size. Because each university sorts it, then the k files must be these universities. All we do is split it for the sorting, and our answer should be $O(n \log k)$.

3. **Suppose the organizer wants to figure out the participants of ranking r among all participants. Given the k sorted files, how does the organizer find the r^{th} largest scores without sorting the scores of all participants? Please describe an algorithm with $O(k(\sum_{i=1}^{k} \log m_i))$ running time and justify its time complexity. Can you do this in $O(\log k(\sum_{i=1}^{k} \log m_i))$ time?**

   I 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.

Prima facie, two good candidates for this task is *counting sort* and *radix sort*. The reason being, both of them sport linear worst-case space complexity. However, due to the numerical nature of the problem and the need for a linear-time worst-case time complexity, lets picks *counting sort* for our problem.

Once we have our elements sorted, it will be a matter of traversing them to find the one which does not have its consecutive adjacent to it.

So what does counting sort need? It needs an *input*, our collection of New York Time crossword puzzles from 01/01/1943 until 12/31/2012. It needs the length of the input, that is n. It needs a number k such that no number can exceed this number in the range. And in terms of space complexity it's going to need a *count array* and an *output array*, along with a few other small variables.

The way you would execute this algorithmically is you would need a hash function which would map the first date in the sequence to zero and the last date in the sequences to the number of dates in the sequence, that is n. Then, you would implement basic counting sort which would accept this n as well as an filled array of crosswords (that is $n-1$ in length). This would sort the crossword in linear time based on it's array with values between 0 and n with all a presumed count of 1 (except our missing one). Then, you would loop over that sorted list and check to see if any date is not followed by it's consecutive date, and you've found our missing one.

Here is a specific instantiation of the general guidelines I outlined above:

```
hash(crossword) {
    // This function will hash 01/01/1943 to 0, 01/02/1943 to 2, ...
    // and 12/31/2012 to n. As it turns out, our n is 25567.

    // Hashing and normalizing date objects is possible in most or all
    // programming languages, and I'm not implementing it here to stay
    // language-agnostic.

    // This is where I would use getDay, getMonth, and getYear.
    // Here is a crude way of describing the needed code:

    if (crossword.getYear() == 1943) {
       if(crossword.getMonth() == 1) {
          if (crossword.getDay() == 1) {
             return 0;
          }
          if (crossword.getDay() == 2) {
             return 1;
          }
```

```
            if (crossword.getDay() == 3) {
               return 2;
            }
            ...
         }
         ...
      }
      ...

      return x;
}

countingSortForCrosswords(crosswords, n) {
   total = 0;   # variable used in algorithm
   output[n];   # the output, that is sorted, array
   count[n];    # used as a histogram
   oldCount[n]; # used to transfer values

   # this is the histogram step, and will be presumably 1
   # except for our missing crossword!

   for x in crosswords {
      count[hash(x)] += 1;
   }

   # this is the starting index step, notice that our maximum
   # in the range is the same as the number of elements

   for (int i = 0; i < n; i++) {
      oldCount = count[i];
      count[i] = total;
      total += oldCount;
   }

   # copy to output array step

   for x in crosswords {
      output[count[hash(x)]] = x;
      count[hash(x)] += 1;
   }

   return output;
}

# returns the crossword before the missing one
```

```
main(n) {
   crosswords[n];
   for (int i = 0; i < n; i++) {
      crosswords[i] = getPuzzle(i);
   }

   countingSortForCrosswords(crosswords, n);

   for (int i = 0; i < n - 1; i++) {
      if (hash(crosswords[i]) != hash(crosswords[i + 1])) {
         return i;
      }
   }
}
```

This will take linear time and require linear space. The reason for it taking linear time is that all the components of our algorithm are linear or constant. The hash function is going to be constant, as good hash functions are. That constant hash function uses only the constant functions that are provided to me. The counting sort algorithm will also be constant, each of my for loops operates a total of n times. Finally, my main function will run all it's for loops in constant time relative to n, where the initialization loop will run as many times as there are crosswords, my sort will be linear, and my check will be linear. That is,

$$\Omega(n + (n + n) + n) = O(n)$$

Furthermore, it will require linear space because the crosswords are cached into an array of size n, the counting sort utilizes some variables and cache arrays of size n, and that's all I use.

$$O(n)$$

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 $O(m \log m + n)$.**

  I don't know.

- **What is the approximation ratio of your algorithm and why?**

  I don't know.