CS 170 HW 2

Due on 2019-02-04, at 10:00 pm

1 Study Group

List the names and SIDs of the members in your study group.

2 Three Part Solution

For each of the algorithm-design questions, please provide a three part solution which includes:

1. The main idea **or** pseudocode underlying your algorithm
2. A proof of correctness
3. A runtime analysis

3 Asymptotic Complexity Comparisons

(a) Order the following functions so that $f_i = O(f_j) \iff i \leq j$. Do not justify your answers.

(i) $f_1(n) = 3^n$
(ii) $f_2(n) = n^{\frac{1}{3}}$
(iii) $f_3(n) = 12$
(iv) $f_4(n) = 2^{\log_2 n}$
(v) $f_5(n) = \sqrt{n}$
(vi) $f_6(n) = 2^n$
(vii) $f_7(n) = \log_2 n$
(viii) $f_8(n) = 2^{\sqrt{n}}$
(ix) $f_9(n) = n^3$

Solution: $f_3, f_7, f_2, f_5, f_4, f_9, f_8, f_6, f_1$

(b) In each of the following, indicate whether $f = O(g)$, $f = \Omega(g)$, or both (in which case $f = \Theta(g)$). ***Briefly*** justify each of your answers. Recall that in terms of asymptotic growth rate, logarithmic $<$ linear $<$ polynomial $<$ exponential.

	$f(n)$	$g(n)$
(i)	$\log_3 n$	$\log_4 n$
(ii)	$n \log(n^4)$	$n^2 \log(n^3)$
(iii)	$\sqrt{n}$	$(\log n)^3$
(iv)	2^n	2^{n+1}
(v)	n	$(\log n)^{\log \log n}$
(vi)	$n + \log n$	$n + (\log n)^2$
(vii)	$\log(n!)$	$n \log n$

Solution:

(i) $f = \Theta(g)$; using the log change of base formula, $\frac{\log n}{\log 3}$ and $\frac{\log n}{\log 4}$ differ only by a constant factor.

(ii) $f = O(g)$; $f(n) = 4n\log(n)$ and $g(n) = 3n^2\log(n)$, and the polynomial in g has the higher degree.

(iii) $f = \Omega((\log n)^3)$; any polynomial dominates a product of logs.

(iv) $f = \Theta(g)$; $g(n) = 2f(n)$ so they are constant factors of each other.

(v) $f = \Omega(g)$; $n = 2^{\log n}$ and $(\log n)^{\log\log n} = 2^{(\log\log n)^2}$, so f grows faster than g since $\log n$ grows faster than $(\log\log n)^2$.

(vi) $f = \Theta(g)$; Both f and g grow as $\Theta(n)$ because the linear term dominates the other.

(vii) $f = \Theta(g)$;
Observe that

$$n! = 1 * 2 * 3 \cdots * n \leq n * n * n \cdots * n \leq n^n$$

and assuming n is even (without loss of generality)

$$n! = 1 * 2 * 3 \cdots * n \geq n * (n-1) * (n-2) \cdots * (n - n/2) \geq \left(\frac{n}{2}\right)^{\frac{n}{2}+1}.$$

Hence $\left(\frac{n}{2}\right)^{\frac{n}{2}} \leq n! \leq n^n$. Then,

$$\frac{n}{2}\log\left(\frac{n}{2}\right) \leq \log(n!) \leq n\log n.$$

and we conclude that the functions grow at the same asymptotic rate.

4 In Between Functions

In this question, we will try to find functions whose rate of growth is strictly between polynomials and exponentials. Specifically, prove or disprove the following claim: If $f : \mathbb{N} \to \mathbb{N}$ is any positive-valued function, then either (1) there exists a constant $c > 0$ so that $f(n) \in O(n^c)$, or (2) there exists a constant $\alpha > 1$ so that $f(n) \in \Omega(\alpha^n)$.

Solution: We will disprove the claim via a counterexample. Let $f(n) = 2^{\sqrt{n}}$. We see that $f(n) \notin \Omega(n^c)$ for any constant $c > 0$. Furthermore, we have $f(n) \in o(\alpha^n)$ for any constant $\alpha > 1$. This implies $f(n) \notin \Omega(\alpha^n)$ for any $\alpha > 1$. Therefore, $f(n)$ does not satisfy either of the two cases of the claim.

As a side note, this shows that there are algorithms whose running time grows faster than any polynomial but slower than any exponential. In other words, there exists a nether between polynomial-time and exponential-time.

5 Bit Counter

Consider an n-bit counter that counts from 0 to $2^n - 1$.

When $n = 5$, the counter has the following values:

Step	Value	# Bit-Flips
0	00000	–
1	00001	1
2	00010	2
3	00011	1
4	00100	3
⋮	⋮	
31	11111	1

For example, the last two bits flip when the counter goes from 1 to 2. Using $\Theta(\cdot)$ notation, find the growth of the *total* number of bit flips (the sum of all the numbers in the "# Bit-Flips" column) as a function of n.

Solution:

The number of times the (i)-th bit from the left is flipped is $\Theta(2^i)$. For example, the first bit is flipped once, and the last is flipped every time, $\Theta(2^n)$ times.

There are n bits, so the total number of bit flips is

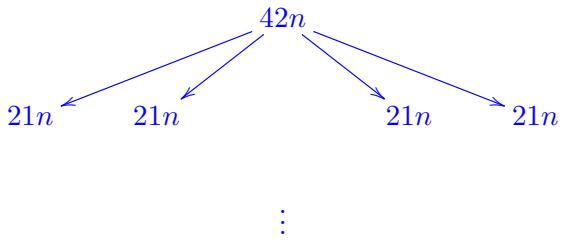
$$\sum_{i=1}^{n}(2^i) = 2^{n+1} - 2 = \Theta(2^n)$$

6 Recurrence Relations

For each part, find the asymptotic order of growth of T; that is, find a function g such that $T(n) = \Theta(g(n))$.

(a) $T(n) = 4T(n/2) + 42n$

Solution: Use the master theorem. Or:



The first level sums to $42n$, the second sums to $84n$, etc. The last row dominates, and we have $\log n$ rows, so we have $42 \cdot 2^{\log n} \cdot n = \Theta(n^2)$.

(b) $T(n) = 4T(n/3) + n^2$

Solution: Use the master theorem (the case $d > \log_b a = \log_3 4$), or expand like the previous question. The answer is $\Theta(n^2)$.

(c) $T(n) = T(\sqrt{n}) + 1$ (You may assume that $T(2) = T(1) = 1$)

Solution: The answer to this question is $\Theta(\log \log n)$. Notice that after the k^{th} step of the recursion, we have the size of the input to be $2^{\log n/2^k}$. Since, the number of recursive levels is $\log \log n$ and each level contributes 1, the solution to the problem is $\Theta(\log \log n)$.

7 Hadamard matrices

The Hadamard matrices $H_0, H_1, H_2, \ldots$ are defined as follows:

- H_0 is the 1×1 matrix $[1]$
- For $k > 0, H_k$ is the $2^k \times 2^k$ matrix

$$H_k = \left[\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array}\right]$$

(a) Write down the Hadamard matrices H_0, H_1, and H_2.

Solution: $H_0 = 1$

$$H_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$H_2 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

(b) Compute the matrix-vector product $H_2 v$, where H_2 is the Hadamard matrix you found above, and $v = \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix}$ is a column vector. Note that since H_2 is a 4×4 matrix, and v is a vector of length 4, the result will be a vector of length 4.

Solution:

$$H_2 v = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 4 \end{bmatrix}$$

(c) Now, we will compute another quantity. Take v_1 and v_2 to be the top and bottom halves of v respectively. Therefore, we have that $v_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$, $v_2 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}$, and $v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$. Compute $u_1 = H_1(v_1 + v_2)$ and $u_2 = H_1(v_1 - v_2)$ to get two vectors of length 2. Stack u_1 above u_2 to get a vector u of length 4. What do you notice about u?

Solution:

$$H_1(v_1 + v_2) = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$H_1(v_1 - v_2) = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ -2 \end{bmatrix} = \begin{bmatrix} 0 \\ 4 \end{bmatrix}$$

We notice that $u = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 4 \end{bmatrix} = H_2 v$

(d) Suppose that

$$v = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

is a column vector of length $n = 2^k$. v_1 and v_2 are the top and bottom half of the vector, respectively. Therefore, they are each vectors of length $\frac{n}{2} = 2^{k-1}$. Write the matrix-vector product $H_k v$ in terms of H_{k-1}, v_1, and v_2 (note that H_{k-1} is a matrix of dimension $\frac{n}{2} \times \frac{n}{2}$, or $2^{k-1} \times 2^{k-1}$). Since H_k is a $n \times n$ matrix, and v is a vector of length n, the result will be a vector of length n.

Solution: $H_k v = \begin{bmatrix} H_{k-1}v_1 + H_{k-1}v_2 \\ H_{k-1}v_1 - H_{k-1}v_2 \end{bmatrix} = \begin{bmatrix} H_{k-1}(v_1 + v_2) \\ H_{k-1}(v_1 - v_2) \end{bmatrix}$

(e) Use your results from (c) to come up with a divide-and-conquer algorithm to calculate the matrix-vector product $H_k v$, and show that it can be calculated using $O(n \log n)$ operations. Assume that all the numbers involved are small enough that basic arithmetic operations like addition and multiplication take unit time.

Solution: Compute the 2 subproblems described in part (d), and combine them as described in part (c). Let $T(n)$ represent the time taken to find $H_k v$. We need to find the vectors $v_1 + v_2$ and $v_1 - v_2$, which takes $O(n)$ time. And we need to find the matrix-vector products $H_{k-1}(v_1 + v_2)$ and $H_{k-1}(v_1 - v_2)$, which take $T(\frac{n}{2})$ time. So, the recurrence relation for the runtime is:

$$T(n) = 2T(\frac{n}{2}) + O(n)$$

Using the master theorem, this give us $T(n) = O(n \log n)$.

8 Fastest Winning Strategy

Suppose a group of n friends, G, are playing a Werewolf-like party game where the group is divided into two sets of people, the citizens and the werewolves. The goal of the citizens is to figure out who the werewolves are or equivalently who the other citizens are. In this game, each person can determine whether someone else is a citizen or a werewolf based on a brief conversation. However, while the citizens will truthfully report whether the person they are paired with is a citizen, the werewolves have no incentive to report the truth. Assuming that there are more than $n/2$ citizens in the group of n people, can you devise efficient algorithms

to determine the identities of the citizens assuming each interaction between people takes $O(1)$ time (Two pairs of people cannot interact at the same time). Please provide a three part solution to both problems.

(a) Show how to solve this problem in $O(n \log n)$ time. (Hint: Split the group of people into two sub-groups of equal size. Can you devise an appropriate divide and conquer algorithm to solve the problem? This should give you an $O(n \log n)$ algorithm.)

(b) (**Extra Credit**) Can you give a linear-time algorithm?

Solution: In both of the solutions below, two friends agreeing means that both of them report that the other is a citizen and them disagreeing means one of them claims the other is a werewolf.

(a) **Main idea** For the divide and conquer approach, we will try to devise an algorithm with the guarantee that if it is given a set of friends A with more than half being citizens, it returns a citizen. To this end, we start by first paritioning the group of friends G into two sub-groups A and B of roughly equal size (If n is odd, one of the sets will have 1 more element than the other) and running the algorithm recursively on each set A and B. Suppose, the candidates returned by the algorithm on A and B are f_1 and f_2. We then ask f_1 and f_2 to interact with each of the people in G and return whoever agrees with more people. In case of a tie, we may return either one.

Proof of correctness By strong induction on n:

Induction hypothesis The algorithm is correct for $k \leq n-1$.

Base Case If $n = 1$, there is only one person in the group who is a citizen and the algorithm is trivially correct.

Induction step: Suppose the algorithm is given as input G with $|G| = n$ and more than half of the friends in G are citizens. Then, after paritioning the group into two grups A and B, at least one of the two groups has citizens as more than half of its members. Let the algorithm return f_1 and f_2 on input A and B respectively. Therefore, at least one of f_1 and f_2 is a citizen. If both are citizens, either one is returned and the algorithm remains correct. Otherwise, more than half of the friends on G will disagree with the werewolf and therefore, we will return the citizen. Thus, the algorithm remains correct in this case.

Running time analysis Two calls to problems of size $n/2$, and then linear time to compare the two people returned to each of the friends in the input group: $T(n) = 2T(\frac{n}{2}) + O(n) = O(n \log n)$.

(b) **Main idea** The main idea is that removing a pair of people one of whom is a werewolf does not change the fact that the citizens are in the majority. As before, we will try to devise an algorithm to output a single citizen. We will maintain a set of candidate citizens which will intially be the entire set of friends G and gradually reduce the size of the set until it contains only a single element. If n is odd, we choose one person, say f_1 from the set of friends and ask them to interact with each of the remaining $n-1$ people. If at least $(n-1)/2$ people agree that f_1 is a citizen, we say f_1 and everyone

who agrees with them is a citizen. Otherwise, we remove f_1 as a candidate citizen and recurse on the remaining $n-1$ people. Therefore, we may assume that n is even. We now pair up people in the group arbitrarily to form $n/2$ groups. Suppose the groups are $(f_1, f_2), \ldots, (f_{n-1}, f_n)$. We ask the two members in each group whether the other member is a citizen. If both report that the other member is a citizen, we retain one of the members arbitrarily. If either member of a pair declares that the other is not a citizen, we remove both of them from our candidate set. Through this process, we obtain a new set G' that is atmost half the size of the original set. We then recurse with the new set G' and return when we are reduced to a set with a single element.

Proof of correctness As before, we will prove the correctness of our algorithm by strong induction on the number of friends playing the game n:

Induction hypothesis Given a set of people of size k where more than half the people are citizens, the algorithm correctly returns a citizen.

Base Case If $n = 1$, the group only contains one friend who is guaranteed to be a citizen.

Induction step If n is odd, we first choose a friend, say f_1 and ask every other friend in the list whether f_1 is a citizen. Recall, we return f_1 as a citizen if at least $(n-1)/2$ people agree that f_1 is a citizen. Otherwise, we remove f_1 from our set of citizen candidates. Now, if f_1 is indeed a citizen, at least $(n-1)/2$ friends would agree that they are a citizen as more than half of the n friends are citizens. In this case, we would correctly return f_1. Otherwise, if f_1 is a werewolf, there are at least $(n+1)/2$ citizens in the set of n friends who would deny that they are a citizen, in which case, we would rightly discard them. Now, the correctness of the algorithm for input size $k = n$ follows from the induction hypothesis as we will recurse on a set of candidates of size $n-1$ where the citizens are still in the majority as we have simply removed a werewolf.

Consider the case where n is even. In this case, we have pairs of the form $(f_1, f_2), (f_3, f_4), \ldots (f_{n-1}, f_n)$. Suppose, we first get rid of all the pairs which one of the friends claims the other is a werewolf. Let (f, g) be such a pair and suppose that f claims that g is a werewolf. Either f is a citizen, in which case g is a werewolf or f is werewolf. In either case, the pair (f, g) contains at least one werewolf. Suppose there are m such pairs. Removing all such pairs, we remove at least m werewolves and at most m citizens. Therefore, among the remaining $n-2m$ candidates, we still have the property that most of the condidates are citizens. Now, in the remaining pairs, we have that each pair consist of either two citizens or two werewolves as in a pair with a citizen and werewolf, the citizen would've reported the other member a werewolf. By arbitrarily picking one member of each pair to remain in our candidate set, we simply reduce the number of candidates by half exactly halving the number of citizens and werewolves forming a new set of size $(n-2m)/2$ where the majority of the people are still citizens. The correctness of the algorithm follows from the induction hypothesis.

Running time analysis In a single run of the algorithm on an input set of size n, we do $O(n)$ work to check whether f_1 is a citizen in the case that n is odd and to pair up the remaining friends and pruning the candidate set to at most $n/2$ people. Therefore,

the runtime is given by the following recursion:

$$T(n) = T\left(\frac{n}{2}\right) + O(n) = O(n).$$