CS 170 Homework 1

Due **9/7/2021, at 10:00 pm (grace period until 11:59pm)**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, you must explicitly write none.

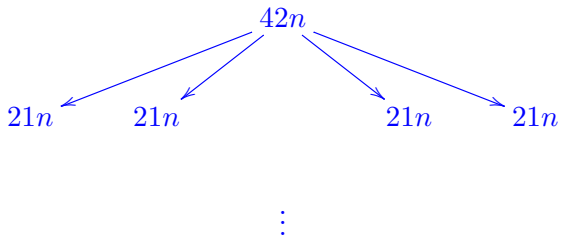
In addition, we would like to share correct student solutions that are well-written with the class after each homework. Are you okay with your correct solutions being used for this purpose? Answer "Yes", "Yes but anonymously", or "No"

2 Recurrence Relations

For each part, find the asymptotic order of growth of T; that is, find a function g such that $T(n) = \Theta(g(n))$. In all subparts, you may ignore any issues arising from whether a number is an integer.

(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 $\Theta(\log \log n)$.

3 Euclid GCD

Euclid's algorithm is an algorithm that computes the greatest common divisor between two integers $a, b \geq 0$:

function EUCLID-GCD(a, b)
  if $a > b$ **then**
    return EUCLID-GCD(b, a)
  else if $a = 0$ **then**
    return b
  else
    return EUCLID-GCD(b **mod** a, a)

As an introduction to the types of things we'll be doing in CS 170, let's analyze this algorithm's runtime and correctness.

(a) Let $n = a+b$. We will try to write this algorithm's runtime as $\mathcal{O}(f(n))$ for some function f, counting every integer operation (including **mod**) as one operation taking constant time. Starting with EUCLID-GCD(a, b) where $a < b$, what will the arguments be after two recursive calls?

Solution:

If we start with EUCLID-GCD(a, b), after one recursive call we have EUCLID-GCD(b **mod** a, a). After two recursive calls, we have EUCLID-GCD(a **mod** (b **mod** a), b **mod** a).

(b) Assume still that $a < b$, and let n' be the sum of a and b after two recursive calls. Show that after two recursive calls, $n' \leq \frac{1}{2}n$. What runtime do we end up with, in the form $\mathcal{O}(f(n))$?

Hint: Use the properties that: (1) b **mod** $a \leq a$, (2) If $b \geq a$, then b **mod** $a \leq b - a$.

Solution:

From part a, we know that a after two recursive calls is a **mod** (b **mod** a), and b after two recursive calls is b **mod** a, so $n' = a$ **mod** (b **mod** a) + b **mod** a.

We can use the first property from the hint to note that $a \geq b$ **mod** a, which will allow us to use the second property from the hint to see that a **mod** (b **mod** a) $\leq a - (b$ **mod** $a)$.

So, $n' = a$ **mod** (b **mod** a) + b **mod** $a \leq a - (b$ **mod** $a) + b$ **mod** $a = a$

Since $a < b$, $n = a + b > 2a$, meaning $n' \leq \frac{1}{2}n$. This means we will need only $\mathcal{O}(\log n)$ recursive calls to complete the whole computation; since each call does only a constant number of operations, this means the runtime is $\mathcal{O}(\log n)$ overall.

> Notice that the base of the log isn't actually important! Because $\log_d x = \frac{\log_c x}{\log_c d}$, all bases of log are related by a constant, which $\mathcal{O}$ ignores.

function EUCLID-GCD(a, b)
  if $a > b$ **then**
    return EUCLID-GCD(b, a)

else if $a = 0$ **then**
  return b
else
  return EUCLID-GCD(b **mod** a, a)

(c) Let's look at correctness. We can prove that this algorithm returns the correct answer using a proof by induction. Note that the result of the algorithm does not depend on if $a > b$ or $a \leq b$, so let's focus on the case that $a \leq b$, and the other case follows.

As our base case, let $a = 0$. Argue that EUCLID-GCD$(0, b)$ returns the correct answer for all b.

Solution: For all integers d, $d \mid 0$. Then the GCD is the largest integer d such that $d \mid b$, which is b. So the EUCLID-GCD acts correctly in this case.

(d) Now for the inductive step: Assume that EUCLID-GCD(a, b) computes the right answer for all $a \leq b \leq k-1$. Show that EUCLID-GCD(a, b) computes the right answer for all $a \leq b = k$.

Hint: Use the property that $d \mid a$ and $d \mid b \iff d \mid a$ and $d \mid (b$ **mod** $a)$.

Solution:

By the property in the hint, the set of common divisors of $\{a, b\}$ is the same as the set of common divisors of $\{b$ **mod** $a, a\}$, so the greatest of these common divisors must also be the same.

> If you are interested in why the property is true, think about the fact $b = \ell \cdot a + (b$ **mod** $a)$ for some ℓ. How does this show $d \mid a$ and $d \mid b \iff d \mid a$ and $d \mid (b$ **mod** $a)$?

It remains to be shown that EUCLID-GCD$(b$ **mod** $a, a)$ is calculated correctly. There are two cases:

Case 1: If $a = b$, then b **mod** $a = 0$. By the base case, EUCLID-GCD$(0, a)$ is calculated correctly.

Case 2: If $a < b$, then $a \leq k-1$. By the inductive hypothesis, EUCLID-GCD$(b$ **mod** $a, a)$ is calculated correctly.

4 Sequences

Suppose we have a sequence of integers A_n, where $A_0, \ldots, A_{k-1} < 50$ are given, and each subsequent term in the sequence is given by some integer linear combination of the k previous terms: $A_i = A_{i-1}b_1 + A_{i-2}b_2 + \cdots + A_{i-k}b_k$. You are given as inputs A_0 through A_{k-1} and the coefficients b_1 through b_k.

(a) Devise an algorithm which computes A_n **mod** 50 in $\mathcal{O}(\log n)$ time (**Hint:** use the matrix multiplication technique from class). You should treat k as a constant, and you may assume that all arithmetic operations involving numbers of $\mathcal{O}(\log n)$ bits take constant time. [1]

Give a 3-part solution as described in the homework guidelines.

Solution:

Algorithm description

For $n < k$, we can simply return A_n from the input as it is given. Henceforth we assume $n \geq k$.

Write

$$B = \begin{bmatrix} b_1 & b_2 & \cdots & b_{k-1} & b_k \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

so that

$$B \begin{bmatrix} A_{i-1} \\ A_{i-2} \\ \vdots \\ A_{i-k+1} \\ A_{i-k} \end{bmatrix} = \begin{bmatrix} A_i \\ A_{i-1} \\ \vdots \\ A_{i-k+2} \\ A_{i-k+1} \end{bmatrix}$$

We can raise B to a power mod 50 using repeated squaring as done in class. Writing

$$\mathbf{a}_0 = \begin{bmatrix} A_{k-1} \\ A_{k-2} \\ \vdots \\ A_0 \end{bmatrix},$$

we return $(B^{n-k+1}\mathbf{a}_0)_1$ **mod** 50.

Proof of correctness

Since repeated squaring is an algorithm from lecture, we can use its correctness without proof; this allows us to say that the algorithm correctly computes $B^n\mathbf{a}_0$ **mod** 50. We will prove by induction that $B^{n-k+1}\mathbf{a}_0 = [A_n, A_{n-1}, \ldots, A_{n-k+1}]^\top$ (which we define as $\mathbf{a}_{n-k+1}$) for any $n \geq k-1$.

Base case: $B^0\mathbf{a}_0 = I\mathbf{a}_0 = \mathbf{a}_0$, and thus the claim is true for $n = k-1$.

[1] A similar assumption – that arithmetic operations involving numbers of $\mathcal{O}(\log N)$ bits take constant time, where N is the number of bits needed to describe the entire input – is known as the *transdichotomous word RAM model* and it is typically at least implicitly assumed in the study of algorithms. Indeed, in an input of size N it is standard to assume that we can index into the input in constant time (do we not typically assume that indexing into an input array takes constant time?!). Implicitly this is assuming that the register size on our computer is at least $\log N$ bits, which means it is natural to assume that we can do all standard machine operations on $\log N$ bits in constant time.

Inductive step: Assume $B^{n-k}\mathbf{a}_0 = \mathbf{a}_{n-k}$; then $B^{n-k+1}\mathbf{a}_0 = BB^{n-k}\mathbf{a}_0 = B\mathbf{a}_{n-k}$. By our construction of B, $B\mathbf{a}_{n-k} = \mathbf{a}_{n-k+1}$.

Runtime analysis From the analysis of the repeated squaring method in class, we know that computing B^{n-k+1} **mod** 50 takes $\mathcal{O}(k^3 \log n) = O(\log n)$ flops. Each flop is constant time since the numbers are always at most 50 and hence of $O(1)$ bit complexity. The rest of the algorithm consists only of multiplying B^n by our vector and selecting an element from it; since k is a constant, this work can be done in constant time and our algorithm takes $\mathcal{O}(\log n)$ time overall.

(b) Devise an even faster algorithm which doesn't use matrix multiplication at all. Once again, you should still treat k as a constant.

Hint: Exploit the fact that we only want the answer mod a constant (here 50).

Give a 3-part solution as described in the homework guidelines.

Solution:

Algorithm description

We will compute consecutive terms of the sequence **mod** 50 (through whatever means we like, e.g. using the previous problem part), starting with A_0 and continuing until the sequence repeats: that is, until we find the first index i for which there is some earlier index j where $A_i = A_j$ **mod** 50, $A_{i+1} = A_{j+1}$ **mod** 50, $A_{i+2} = A_{j+2}$ **mod** 50, etc., through $A_{i+k-1} = A_{j+k-1}$ **mod** 50. Then $A_n = A_{(n-j) \bmod (i-j)+j}$ **mod** 50; we have already computed $A_{(n-j) \bmod (i-j)+j}$ (since $(n-j) \bmod (i-j) < i-j$), so we can return it.

Proof of correctness

Note that each term of the sequence depends only on the k previous terms, and we are computing our sequence **mod** 50, there are only 50^k settings to any run of k terms in the sequence. So, our sequence must eventually be periodic, and it must start repeating somewhere in the first 50^k terms. We detect the period of the seqence as $(i-j)$, so after adjusting for the index j at which our sequence first fell into the repeating pattern, we have that $(n-j) \bmod (i-j) + j$ is the equivalent position in the repeating sequence to n.

Runtime analysis

The only work this algorithm does is to compute at most $50^k = O(1)$ terms of the sequence, and do a constant amount of arithmetic involving indices (none larger than n). So, this is a constant time algorithm!

5 Decimal to Binary

Given the n-digit decimal representation of a number, converting it into binary in the natural way takes $O(n^2)$ steps. Give a divide and conquer algorithm to do the conversion and show that it does not take much more time than Karatsuba's algorithm for integer multiplication.

Just state the main idea behind your algorithm and its runtime analysis; no proof of correctness is needed as long as your main idea is clear.

Solution: Similar to Karatsuba's algorithm, we can write x as $10^{n/2} \cdot a + b$ for two $n/2$-digit numbers a, b. The algorithm is to recursively compute the binary representations of $10^{n/2}, a$, and b. We can then compute the binary representation of x using one multiplication and one addition.

The multiplication takes $O(n^{\log_2(3)})$ time, the addition takes $O(n)$ time. So the recurrence we get is $T(n) = 3T(n/2) + O(n^{\log_2(3)})$. This has solution $O(n^{\log_2(3)} \log n)$.

(There is an $O(n^{\log_2(3)})$-time algorithm: We can compute the binary representation of 10^n in time $O(n^{\log_2(3)})$ by doing the multiplications $10 * 10, 10^2 * 10^2, 10^4 * 10^4 \ldots$ - note that the multiplication of $10^{n/2} * 10^{n/2}$ dominates the total runtime of these multiplications. This gives the recurrence $T(n) = 2T(n/2) + O(n^{\log_2(3)})$ instead, with solution $T(n) = O(n^{\log_2(3)})$.)