CSCI 0500: Data Structures, Algorithms, and Intractability (Fall 2025)

Assignment 2

Due at 11:59pm ET, Tuesday, Oct 14

1. (1 point) The following theorem provides tight asymptotic bounds for many recurrences that arise in the analysis of divide-and-conquer algorithms. There is a more general version of the theorem, but the following simple version is sufficient in most cases.

   Theorem 1 (Master Theorem)**.** *Fix an integer $a \geq 1$ and real numbers $b > 1$, $c \geq 0$, and $d > 0$. Consider the following recurrence:*

   $$T(n) = \begin{cases} a \cdot T\left(\frac{n}{b}\right) + n^c & \text{if } n > 1, \\ d & \text{if } n = 1. \end{cases}$$

   Assume that n is a power of b. Then,

   $$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } c < \log_b a, \\ \Theta(n^c \log n) & \text{if } c = \log_b a, \\ \Theta(n^c) & \text{if } c > \log_b a. \end{cases}$$

   You will prove this theorem and then use it to solve some recurrences. Note that

   $$\begin{aligned} T(n) &= a \cdot T\left(\tfrac{n}{b}\right) + n^c \\ &= a \cdot \left(a \cdot T\left(\tfrac{n}{b^2}\right) + \left(\tfrac{n}{b}\right)^c\right) + n^c \\ &= a^2 \cdot T\left(\tfrac{n}{b^2}\right) + \tfrac{a}{b^c} \cdot n^c + n^c \\ &= a^3 \cdot T\left(\tfrac{n}{b^3}\right) + \left(\tfrac{a}{b^c}\right)^2 \cdot n^c + \tfrac{a}{b^c} \cdot n^c + n^c \\ &= a^{\log_b n} \cdot T(1) + n^c \cdot \sum_{i=0}^{(\log_b n)-1} \left(\tfrac{a}{b^c}\right)^i \\ &= \Theta(n^{\log_b a}) + n^c \cdot \sum_{i=0}^{(\log_b n)-1} \left(\tfrac{a}{b^c}\right)^i \end{aligned}$$

   where the last step uses that $a^{\log_b n} = n^{\log_b a}$ and $T(1) = d = \Theta(1)$.

   (a) Complete the proof of Theorem 1.

   (Hint: Let $r = \frac{a}{b^c}$. The second term is n^c times the sum of a geometric series $\sum_{i=0}^{(\log_b n)-1} r^i$. You can analyze the three cases: $r < 1$, $r = 1$, and $r > 1$, and derive a tight asymptotic bound for this sum in each case.)

(b) Use Theorem 1 to provide a tight asymptotic bound for each recurrence below. You can assume that n is a power of 2 and $T(1) = 1$.

i. $T(n) = T(n/2) + n$.
ii. $T(n) = 2\,T(n/2) + n$.
iii. $T(n) = T(n/2) + 1$.
iv. $T(n) = 3\,T(n/2) + n$.
v. $T(n) = 7\,T(n/2) + n^2$.
vi. $T(n) = 2\,T(n/2) + 1$.

2. (1 point) In this question, we study algorithms for computing the greatest common divisor and the modular multiplicative inverse.

(a) Given two integers $a \geq b > 0$, the Euclidean algorithm returns their greatest common divisor $\gcd(a, b)$:

```
def gcd(a, b):
    while b != 0:
        (a, b) = (b, a % b)
    return a
```

Prove that on input $a \geq b > 0$, the algorithm uses $O(\log a)$ modulo operations in the worst case.

(Hint: Consider how much a decreases after two iterations.)

(b) Given two integers $a \geq b > 0$, the extended Euclidean algorithm returns three integers (d, x, y) such that $ax + by = d = \gcd(a, b)$.

```
def extended_gcd(a, b):
    if b == 0:
        return (a, 1, 0)
    else:
        (d, x1, y1) = extended_gcd(b, a % b)
        x = y1
        y = x1 - (a // b) * y1
        return (d, x, y)
```

Prove the correctness of `extended_gcd(a, b)` using mathematical induction.

Remark. Let n and a be integers with $0 < a < n$. A multiplicative inverse of a modulo n is an integer y such that $ay \equiv 1 \pmod{n}$, which exists if and only if $\gcd(a, n) = 1$. A multiplicative inverse can be computed by running the extended Euclidean algorithm on input (n, a), which returns $(1, x, y)$ with $nx + ay = 1$. It follows that $ay \equiv 1 \pmod{n}$.

3. (1 point) In class, we discussed how repeated squaring can speed up (modular) exponentiation. In this question, we explore another application of repeated squaring.

   The Fibonacci sequence is defined as:

$$F_n = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ F_{n-1} + F_{n-2} & \text{if } n \geq 2. \end{cases}$$

   Given F_k and F_{k+1}, the next number F_{k+2} can be computed using matrix multiplication:

$$\begin{pmatrix} F_{k+2} \\ F_{k+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{k+1} \\ F_k \end{pmatrix}$$

   By mathematical induction, this implies

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}.$$

   Based on the above equation, design an algorithm `fibonacci(n)` that returns F_n.

   You can use the function `matrix_mult(A, B)`, which returns the product of two 2×2 matrices A and B, without implementing it. For one call `fibonacci(n)`, your algorithm should make $O(\log n)$ calls to `matrix_mult`. (You do not need to analyze the runtime of your algorithm.)