

CSCI 0500: Data Structures, Algorithms, and Intractability (Fall 2026)

Assignment 2

Due at 11:59pm ET, Friday, Oct 2

- (1 point) The following theorem provides tight asymptotic bounds for many recurrences that arise in the analysis of divide-and-conquer algorithms.

Theorem 1 (Master Theorem). *Fix an integer $a \geq 1$ and real numbers $b > 1$, $c \geq 0$, and $d \geq 0$. Let $f(n)$ be a positive function with $f(n) = \Theta(n^c)$. Consider the following recurrence:*

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

Assume that n is a power of b . Then,

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

In this question, you will prove this theorem and then use it to solve some recurrences.

Expanding the recurrence for $\log_b n$ levels gives

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

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

Using $f(n) = \Theta(n^c)$, we have

$$T(n) = d \cdot n^{\log_b a} + \Theta\left(\sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^c\right) = d \cdot n^{\log_b a} + n^c \cdot \Theta\left(\sum_{i=0}^{\log_b n - 1} \left(\frac{a}{b^c}\right)^i\right).$$

- (a) Complete the proof of Theorem 1. (Hint: Let $r = \frac{a}{bc}$. Analyze the geometric sum in the three cases $r > 1$, $r = 1$, and $r < 1$.)
- (b) Use Theorem 1 to give a tight asymptotic bound for each recurrence below. You can assume that n is a power of 2 and $T(1) = 1$.
- $T(n) = T(n/2) + 3n$.
 - $T(n) = 2T(n/2) + n$.
 - $T(n) = T(n/2) + 1$.
 - $T(n) = 3T(n/2) + n$.
 - $T(n) = 7T(n/2) + n^2$.
 - $T(n) = 2T(n/2) + 1$.

Remark. In analyzing algorithms, we sometimes have only an upper bound on the runtime. In the setting of Theorem 1, if the recurrence for $n > 1$ is replaced by $T(n) \leq a \cdot T(\frac{n}{b}) + O(n^c)$, then the conclusion holds with O instead of Θ . You are encouraged to prove this, but it is not required for this assignment.

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

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

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

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

```
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 $\text{gcd}(a, n) = 1$. When $\text{gcd}(a, n) = 1$, running the extended Euclidean algorithm on input (n, a) 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:

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

In this question, let the computational cost (i.e., runtime) be the total number of additions and multiplications used by the algorithm.

Design an algorithm `fibonacci(n)` that returns F_n for $n \geq 0$ with runtime $O(\log n)$. Briefly justify the runtime of your algorithm.

Hint. The recurrence of the Fibonacci sequence can be written as:

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

By mathematical induction, this implies

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

You can use `matrix_mult(A, B)` to multiply two 2×2 matrices without implementing it. Each call to `matrix_mult` uses a constant number of additions and multiplications.

Remark. What is the runtime of your algorithm in the bit-cost model? You are encouraged to think about this, but it is not required for this assignment.