DS5012: Foundations of Computer Science
Module 1 Homework: Analysis of Algorithms

Tom Lever (tsl2b@virginia.edu)

May 30, 2025

Q1 For each function below, state its Big-O classification.

a. $f(n) = 2n^2 + 3n + 1$. As n grows, the quadratic term n^2 dominates. The function is quadratic. $f(n) \in O(n^2)$.

b. $g(n) = n \log(n) + 5n$. As n grows, the log-linear term $n \log(n)$ dominates. $g(n) \in O[n \log(n)]$.

c. $h(n) = 3 \cdot 2^n + n^3$. As n grows, the exponential term 2^n dominates. $h(n) \in O\left(2^n\right)$.

Q2 True or False? "Big-O notation describes an exact growth rate." False. Big-O notation describes an upper bound to a growth rate. If an algorithm $f(n) \in O[g(n)]$, then $f(n)$ grows no faster than $cg(n)$ for some positive constant c for all n greater than or equal to some positive constant n_0. Big-Θ notation describes a tight bound and both a lower bound and an upper bound to a growth rate. If an algorithm $f(n) \in \Theta[g(n)]$, then $f(n)$ grows no slower than $c_1 g(n)$ and no faster than $c_2 g(n)$ for some positive constants c_1 and c_2 for all n greater than some positive constant n_0.

Q3 Analyze the following code and give its time complexity in Big-O form. Show your work.

```
for i in range(n):
    for j in range(i):
        print(i, j)
```

The inner for loop is executed n times. The print statement runs $0, 1, ..., n-2, n-1$ times, where there are n terms. The first and ultimate terms, second and penultimate terms, and other pairs of terms may be added to provide $n/2$ terms all equal to $n-1$. The print statement runs $n/2(n-1)$ times. The print statement has time complexity $O(1)$. The algorithm has time complexity $n/2(n-1)O(1) = O[n/2(n-1)1] = O(n^2)$.

Q4 Solve the recurrence

$$T(n) = 2T(n/2) + n$$

using the Master Theorem. State which case applies and the resulting complexity.

Recurrences allow finding upper bounds of time complexities of recursive algorithms.

The time cost $T(n)$ of an algorithm is a function of the number of elements n in an input representing the number of operations of the algorithm.

A recurrence is an equation that expresses the time cost $T(n)$ of an algorithm in terms of the time cost $T(p)$, a function of the number of elements p in a smaller input of the same algorithm.

The Master Theorem method is a cookbook approach for solving recurrences strictly of the form $T(n) = aT(n/b) + f(n)$ where $a \geq 1$, $b > 1$, and $f(n)$ represents the number of non-recursive splitting and combining operations of the algorithm. The method compares the function f with the critical function $g(n) = n^{\log_b(a)}$.

In our case, $a = 2$, $b = 2$, and $f = n$. The critical function $g = n^{log_2(2)} = n^1 = n$. "Case 2: $f(n)$ is 'Balanced' (Work Spread Evenly)" applies. f grows at the same rate as g. Formally, $f \in \Theta(g)$. The complexity of our recurrence is $T(n) = \Theta[g \log(n)] = \Theta[n \log(n)]$.

Q5 Algorithm A runs in $100n \log_2(n)$ time. Algorithm B runs in n^2 time. Find the smallest integer n_0 such that for all $n \geq n_0$, Algorithm A is faster than Algorithm B; i.e., $100n \log_2(n) < n^2$.

$f(n) = 100n \log_2(n)$ is less than $g(n) = n^2$ for n between 0 and slightly more than 1. Because f is greater than g for a while after that, we cannot choose 0 or 1 as n_0. We need to find the next intersection between the two time functions. I created columns n, $f(n) = 100n \log_2(n)$, and $g(n) = n^2$ in Excel. For $n = 996$, $f = 100 \cdot 996 \log_2(996) = 992,016.192$ and $g = 992,016$. f is still greater than g. For $n = 997$, $f = 100 \cdot 997 \log_2(997) = 993,156.534$ and $g = 994,009$. g is now greater than f. $n_0 = 997$.

Q6 Analyze the time complexity of this linear-regression function. Assume 'X' is an $n \times d$ matrix (list of lists) and 'y' is a vector (list) of length n.

```
import numpy as np

def linear_regression_loops(X, y):
    '''
    n is the number of samples.
    d is the number of features.
    X is an n x d matrix and may be thought of as a list of lists.
    y is a vector of length n.
    '''

    n = len(X) # O(1)
```

```
d = len(X[0]) # O(1)

# 1. Transpose X: XT is d x n

XT = [[0 for _ in range(n)] for _ in range(d)]
for i in range(n):
    for j in range(d):
        XT[j][i] = X[i][j]

# 2. Matrix Multiplication: XT * X -> XT_X (XT_X is d x d)
XT_X = [[0 for _ in range(d)] for _ in range(d)]
for i in range(d):
    for j in range(d):
        sum_val = 0
        for k in range(n):
            sum_val += XT[i][k] * X[k][j]
        XT_X[i][j] = sum_val

# 3. Matrix Inversion: inv(XT_X) -> inv_XT_X (d x d)
inv_XT_X = np.linalg.inv(XT_X)

# 4. Compute intermediate term: XT * y -> XT_y (d x 1)
XT_y = [0 for _ in range(d)]
for i in range(d): # Row of result (d)
    sum_val = 0
    for j in range(n): # Common dimension (n)
        sum_val += XT[i][j] * y[j]
    XT_y[i] = sum_val

# 5. Final Multiplication: inv_XT_X * XT_y -> w (d x 1)
w = [0 for _ in range(d)]
for i in range(d):
    sum_val = 0
    for j in range(d):
        sum_val += inv_XT_X[i][j] * XT_y[j] #
    w[i] = sum_val

return w
```

Give the overall time complexity in terms of n (samples) and d (features), based on the explicit loops and the stated complexity for matrix inversion. Identify the dominant term(s).

Block 0
Defining number of samples n has time complexity $O(1)$.
Accessing an element of a list has time complexity $O(1)$. Because of this, we

don't consider the time complexity of accessing an element further.

Defining number of features involves finding the length of the list with time complexity $O(1)$.

The time complexity of Block 0 is $O(1)$.

Block 1

According to StackOverflow, function range has time complexity $O(1)$. Because of this, we don't consider the time complexity of range further.

Accessing the element of a list within a list has time complexity $O(1+1) = O(1)$. Because of this, we don't consider the time complexity of accessing an element further.

Initializing the transpose of X involves iterating d times and for each iteration iterating n times. Assigning 0 has a time complexity of $O(1)$. Initializing has a time complexity of $O(dn)$.

Assigning values to the transpose involves iterating n times and for each iteration iterating d times. Assigning a value has a time complexity of $O(1)$. Assigning values has a time complexity $O(nd) = O(dn)$.

The time complexity of Block 1 is $O(dn)$.

Block 2

Initializing the product $P = X^T X$ involves iterating d times and for each iteration iterating d times. Assigning 0 has a time complexity of $O(1)$. Initializing has a time complexity $O\left(d^2\right)$.

Assigning values to the product involves

1. for Level 1, iterating d times,
2. for Level 2, iterating d times,
3. for Level 3, assigning 0, iterating n times, and assigning a value, and
4. for Level 4, calculating a scalar product, adding values, and assigning a value.

Assigning values to the product has a time complexity $O[dd(1+n+1)(1+1+1)] = O\left(d^2 n\right)$.

The time complexity of Block 2 is $O\left(d^2 + d^2 n\right) = O\left(d^2 n\right)$.

Block 3

According to Wikipedia, if we assume that numpy performs Gauss-Jordan elimination, then matrix inversion has a time complexity $O\left(d^3\right)$.

The time complexity of Block 3 is $O\left(d^3\right)$.

Block 4

Initializing the product $Q = X^T y$ involves iterating d times. Assigning 0 has a time complexity of $O(1)$. Initializing has a time complexity $O\left(d\right)$.

Assigning values to the product involves

1. for Level 1, iterating d times,
2. for Level 2, assigning 0, iterating n times, and assigning a value, and
3. for Level 3, calculating a scalar product, adding values, and assigning a value.

Assigning values to the product has a time complexity $O[d(1+n+1)(1+1+1)] = O\left(dn\right)$.

Block 5

Initializing the product $R = P^{-1}Q$ involves iterating d times. Assigning 0 has a time complexity of $O(1)$. Initializing has a time complexity $O(d)$.

Assigning values to the product involves

1. for Level 1, iterating d times,
2. for Level 2, assigning 0, iterating d times, and assigning a value, and
3. for Level 3, calculating a scalar product, adding values, and assigning a value.

Assigning values to the product has a time complexity $O[d(1 + d + 1)(1 + 1 + 1)] = O(d^2)$.

Block 5 has a time complexity $O(d + d^2) = O(d^2)$.

Block 6

Returning a value has time complexity $O(1)$.

Block 6 has time complexity $O(1)$.

The time complexity of the linear regression function is

$$O\left(1 + dn + d^2n + d^3 + dn + d^2 + 1\right)$$
$$O\left(1 + dn + d^2 + d^2n + d^3\right)$$
$$O\left(d^3 + d^2n\right)$$

.

d^2n grows faster than and dominates d^2, dn, and 1.

If we were strictly considering the time complexity based on the explicit loops and matrix inversion, then the overall complexity would be

$$O\left(dn + d^2n + d^3 + dn + d^2\right)$$
$$O\left(dn + d^2 + d^2n + d^3\right)$$
$$O\left(d^3 + d^2n\right)$$

d^2n grows faster than and dominates d^2 and dn.

The dominant terms are a cubic function of the number of features and a function equal to the product of a quadratic function of the number of features a linear function of the number of samples. When the number of samples is greater than the number of features, $d^2n > d^3$, and the former term grows faster than the latter term and dominates. When the number of features is greater than the number of samples, $d^3 > d^2n$, and the former term grows faster than the latter term and dominates. d^2n is associated with calculating product P. d^3 is associated with inversion.

Q7 Consider the following recursive function. Analyze its time complexity. Let n be the size of the input list 'data'.

```
def recursive_mystery(data):
    n = len(data)
    # Base case
    if n <= 1:
        return 1

    # Recursive step: Divide into 3 parts (approx n/3)
    size_third = n // 3
    part1 = data[0 : size_third]
    part2 = data[size_third : 2 * size_third]
    part3 = data[2 * size_third : n]

    # Make 3 recursive calls
    res1 = recursive_mystery(part1)
    res2 = recursive_mystery(part2)
    res3 = recursive_mystery(part3)

    # Combine step:
    count = 0
    for i in range(n):
        # Performing some constant time work per element
        count += (i + res1 + res2 + res3) % 100

    return count
```

Set up a recurrence relation for the time complexity $T(n)$ of this function, considering the cost of slicing. Solve the recurrence relation (e.g., using the Master Theorem) and state the Big-O time complexity.

Recurrences allow finding upper bounds of time complexities of recursive algorithms.

The time cost $T(n)$ of an algorithm is a function of the number of elements n in an input representing the number of operations of the algorithm.

A recurrence is an equation that expresses the time cost $T(n)$ of an algorithm in terms of the time cost $T(p)$, a function of the number of elements p in a smaller input of the same algorithm.

In the base case, when n is 0 or 1, the algorithm simply defines n, checks whether the base case is relevant, and returns 1. The base case has time complexity $\Theta(1)$.

The cost of slicing data into `part1` involves creating a new list with the first third elements in data or references to those elements. Preallocating a list would have time complexity $\Theta(n/3)$. Accessing elements or elements would have time complexity $\Theta(n/3)$. Storing elements or references would have time complexity $\Theta(n/3)$. Slicing data into `part1` would have time complexity $\Theta(n/3)$.

Slicing data into `part2` would have time complexity $\Theta(n/3)$.

Slicing data into `part3` would have time complexity $\Theta(n/3)$ or $\Theta(n/3+1)$.

Slicing has a time complexity $\Theta(n)$ and a time cost $g(n) = n$.

Within recursive mystery, recursive mystery is called 3 times. Each time, a part is passed to recursive mystery. The number of recursive calls $a = 3$. The general size of each call $p = n/3$.

In the combine step, a count is defined with time complexity $\Theta(1)$. The time complexity of the for loop is $\Theta(n)$. During each iteration, an index and 3 integers are added together, the remainder of division by 100 is found, the count is accessed, the count is increased, and the count is recorded. The time complexity of the guts of the for loop is $\Theta(1)$. The combine step has time cost $h(n) = n$.

The non-recursive time cost to split and combine is $f(n) = n$.

Our recurrence is $T(n) = aT(p) + f(n) = aT(n/b) + f(n) = 3T(n/3) + n$.

The Master Theorem method is a cookbook approach for solving recurrences strictly of the form $T(n) = aT(n/b) + f(n)$ where $a \geq 1$, $b > 1$, and $f(n)$ represents the number of non-recursive splitting and combining operations of the algorithm. The method compares the function f with the critical function $k(n) = n^{\log_b(a)}$.

In our case, $a = 3$, $b = 3$, and $f = n$. The critical function $g = n^{log_3(3)} = n^1 = n$. "Case 2: $f(n)$ is 'Balanced' (Work Spread Evenly)" applies. f grows at the same rate as k. Formally, $f \in \Theta(k)$. The complexity of our recurrence is $T(n) = \Theta[k \log(n)] = \Theta[n \log(n)]$.

Q8 Amortized analysis: dynamic-array append. Consider the "append" method shown below.

```
class DynamicArray:
    def __init__(self):
        self._capacity = 1
        self._n = 0 # number of actual elements
        self._A = [None] * self._capacity

    def _resize(self, new_capacity):
        # Create new array B
        B = [None] * new_capacity

        # Copy elements from A to B
        for i in range(self._n):
            B[i] = self._A[i]

        # Update array reference and capacity
        self._A = B
        self._capacity = new_capacity

    def append(self, element):
        if self._n == self._capacity:
            # Resize to double capacity when full
            new_capacity = 2 * self._capacity
            self._resize(new_capacity)

        # Append element (after potential resize)
        self._A[self._n] = element
        self._n += 1
```

Explain the difference between the worst-case cost of a single "append" operation and the amortized cost over a sequence of operations. Explain why the amortized cost is $O(1)$.

The worst-case cost of a single "append" operation occurs when the number of elements already in the dynamic array equals the capacity of the dynamic array, append is called, and the dynamic array resizes to double its previous capacity. In this case, the cost of append depends on the cost of `_resize`. The high level task of method `_resize` is copying an element from the existing internal array to a new internal array. Because there are n elements, there are n high level tasks in the worst case. The worst-case cost is n. The worst case time complexity is $O(n)$.

An amortized "append" cost is an average "append" operation cost over a sequence of "append" operations. Even when some "append" operations are occasionally expensive, the average cost per operation may still be low. Our dynamic array performs cheap "append" operations most of the time and expensive "append" operations occasionally.

We can determine an amortized "append" cost by using an aggregate method and dividing the total cost of a sequence of "append" operations by the number of "append" operations; i.e., by finding the average cost of a sequence of "append" operations. This is more complex than finding the worst-case cost of a single "append" operation. Alternatively, we can use an accounting method and assign artificial costs to operations.

The amortized cost over a sequence of "append" operations is an average of the sum of a number of 1s corresponding to actual adds and a number of copies corresponding to resizing. In the first n appends, there are n actual adds. If we append elements so that we have $n = 32+5 = 37$ elements in our array, then the sum of copies corresponding to resizing is $C = 1+2+4+8+16+32 = 64-1 = 63$, which is less than $2n = 74$. In general, if we append elements so that we have $n = 2^{p-1} + q$ elements in our array, then the sum of copies corresponding to resizing is $1 + 2 + 4 + ... + 2^{p-2} + 2^{p-1} + q = 2^p - 1 + q$, which is less than $2n = 2\left(2^{p-1} + q\right) = 2 \cdot 2^{p-1} + 2q = 2^p + 2q$. The sum of actual adds and new copies corresponding to resizing is approximately $n + 2n$. The amortized cost over n appends is approximately $(n + 2n)/n = 1 + 2 = 3$. The amortized cost over n appends is approximately constant. The amortized time complexity over n appends is $O(1)$.

Although some operations take $O(n)$ time, they occur so infrequently that the total cost over many operations grows linearly, and the amortized cost does not grow.