Mohammed VI Polytechnic University
Institute of Science, Technology & Innovation
Al-Khwarizmi Department
Africa Business School

Assignment 1: Complexity Analysis

M123: Numerical Linear Algebra & Parallel Computing

Submitted by:
Meriem ELKHAL

Instructors:
Nourredine OUHADOU
Mohamed Jalal MAAOUNI
Ibrahim EL MOUNTASSER

March 19, 2023

Introduction

Given an integer n, count the number of its divisors

Solution 1:

```
def count_divisors(n):
    count = 0
    d = 1
    while d <= n:
        if n % d == 0:
            count += 1
        d += 1
    return count
```

Solution 2:

```
def count_divisors(n):
    count = 0
    d = 1
    while d * d <= n:
        if n % d == 0:
            count += 1 if n / d == d else 2
        d += 1
    return count
```

1. Solution 1: the function count_divisors counts the number of divisors of a given integer n by iterating over all possible divisors and incrementing a counter variable each time a divisor is found. The loop runs "n" times for each value of "n". So the number of operations performed is directly proportional to the value of "n". Therefore, we can conclude that the algorithm has a time complexity of $O(n)$.
2. Solution 2: the function counts the number of divisors of a given integer n by iterating over all possible divisors up to $\sqrt{n}$ and incrementing a counter variable each time a divisor is found, depending on whether it is a factor of n once or twice. The number of operations is proportional to the square root of n. This can be expressed as $O(\sqrt{(}n))$.
3. We can use the **timeit.timeit** function to time the execution of the two solutions:

Value of **n**	Algorithm **1** (seconds)	Algorithm **2** (seconds)
100	0.0000042	0.0000012
1000	0.0003	0.0000023
10000	0.002	0.0000023
100000	0.0346	0.00030
1000000	0.29	0.0001

Big-O Notation

1. $T(n) = 3n^3 + 2n^2 + \frac{1}{2}n + 7$ let's prove that T(n) = O (n^3)

   For $n \geq 1$, we have:

$$T(n) \leq 3n^3 + 2n^3 + \frac{1}{2}n^3 + 7n^3 = \frac{15}{2}n^3 + 7$$

   Let $c = \frac{15}{2}$ and $n_0 = 1$. Then for all $n \geq 1$ we have:

$$T(n) \leq \frac{15}{2}n^3 + 7 \leq \frac{15}{2}n^3 = cn^3$$

Finally, $T(n) = O(n^3)$.

2. To prove that $\forall k \geq 1, n^k$ is not $O(n^{k-1})$, we need to prove the negation of this statement, which is that $\forall c > 0$ and $\forall n_0 \in \mathbb{N}$, there exists $n \geq n_0$ such that $n^k > cn^{k-1}$. Let $c > 0$ and $n_0 \in \mathbb{N}$ be arbitrary. We want to find a value of n that satisfies the inequality $n^k > cn^{k-1}$.

   Dividing both sides by n^{k-1} gives:

$$n > c$$

   Since $c > 0$, we can choose $n = \max n_0, \lceil c \rceil$. Then we have:

$$n \geq \lceil c \rceil > c \geq \frac{c}{n^{k-1}}$$

   Multiplying both sides by n^{k-1} gives:

$$n^k > cn^{k-1}$$

   Therefore, we have shown that for any $k \geq 1$, n^k is not $O(n^{k-1})$.

Merge sort

1. The function below compares two input arrays using pointers i and j. It adds the smaller value to the output array and moves the pointer of that array forward. If the values are equal, it adds both and moves both pointers forward. Finally, it adds any remaining elements from the arrays to the output array, which is sorted.

```
def merge(A, B):
    i = j = 0
    C = []
    while i < len(A) and j < len(B):
        if A[i] < B[j]:
            C.append(A[i])
            i += 1
        elif A[i] > B[j]:
            C.append(B[j])
            j += 1
        else:
            C.append(A[i])
            C.append(B[j])
            i += 1
            j += 1
    while i < len(A):
        C.append(A[i])
        i += 1
    while j < len(B):
        C.append(B[j])
```

```
        j += 1
    return C
```

2. The time complexity of this function is O(m + n), where m and n are the lengths of the input arrays A and B.

The master method

1. Using the master method, the time complexity of merge sort can be expressed as:

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

where the $O(n)$ term represents the time taken to merge the two halves. Applying the master method, we get:

$a = 2$, $b = 2$, and $f(n) = O(n)$

Therefore, the time complexity of merge sort is $\Theta(n \log n)$.

2. Using the master method, the time complexity of binary search can be expressed as: $T(n) = T\left(\frac{n}{2}\right) + O(1)$

where $O(1)$ represents the constant time taken to compare the middle element with the search element. Applying the master method, we get:

$a = 1$, $b = 2$, and $f(n) = O(1)$

Therefore, the time complexity of binary search is $\Theta(\log n)$.

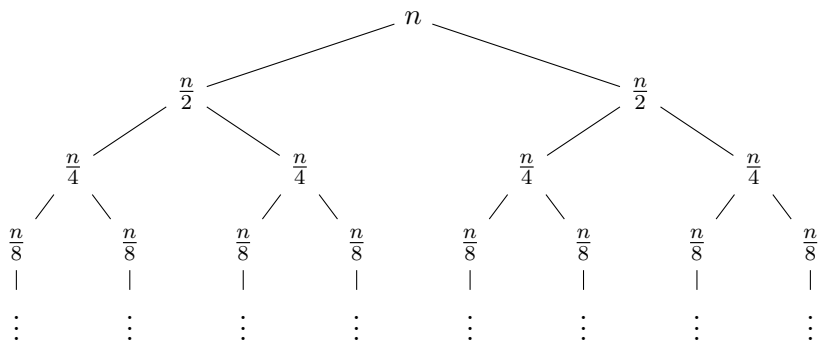
Bonus

1. Let's write a function called merge sort that takes two arrays as parameters and sort those two arrays using the merge sort algorithm.

```
def merge_sort(arr1, arr2):
    if len(arr1) <= 1 and len(arr2) <= 1:
        return sorted(arr1 + arr2)
    mid1 = len(arr1) // 2
    mid2 = len(arr2) // 2
    left1 = arr1[:mid1]
    right1 = arr1[mid1:]
    left2 = arr2[:mid2]
    right2 = arr2[mid2:]
    sorted_left = merge_sort(left1, left2)
    sorted_right = merge_sort(right1, right2)
    return merge(sorted_left, sorted_right)

def merge(left, right):
    result = []
    while left and right:
```

```
        if left[0] <= right[0]:
            result.append(left.pop(0))
        else:
            result.append(right.pop(0))
    if left:
        result.extend(left)
    if right:
        result.extend(right)
    return result

A = [4, 2, 1, 6, 8]
B = [3, 7, 5, 9]
sorted_array = merge_sort(A, B)
print(sorted_array)

Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

2. Let's analyse the complexity of our algorithm without using the master theorem.



In the tree we have:

- At the first level, there is one node with value n.
- At the second level, there are two nodes with value $\frac{n}{2}$.
- At the third level, there are four nodes with value $\frac{n}{4}$.
- $\vdots$
- At the k-th level, there are 2^k nodes with value $\frac{n}{2^k}$.
- The last level will have nodes with value 1, and there will be n of them.

We can compute the total number of nodes in the tree as follows:

$$1 + 2 + 4 + \cdots + 2^{k-1} + n = 2^k - 1 + n$$

where k is the number of levels in the tree. Since the height of the tree is k, we have:

$$2^k - 1 + n = n \implies 2^k = n + 1 \implies k = \log_2(n + 1)$$

Therefore, the tree has $\log_2(n + 1)$ levels.

To compute the time complexity of merge sort based on this tree, we can start at the bottom of the tree and work our way up. At each level k, the total work done is $O(n)$ (since there are n nodes at that level, each of which takes constant time to sort). Therefore, the total work done at all levels is:

$$O(n \log_2(n+1))$$

Therefore, the time complexity of merge sort is $O(n \log n)$.

3. Proof of the 3 cases of the master theorem:

   The Master Method:

   If $T(n) \leq aT\left(\frac{n}{b}\right) + O\left(n^d\right)$ then

$$T(n) = \begin{cases} O\left(n^d \log n\right) & \text{if } a = b^d \quad \text{(Case 1)} \\ O\left(n^d\right) & \text{if } a < b^d \quad \text{(Case 2)} \\ O\left(n^{\log_b a}\right) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

   - Proof of case 1:

$$\begin{aligned} T(n) &= aT\left(\frac{n}{b}\right) + f(n) \\ &\leq a(c(\frac{n}{b})^d \log \frac{n}{b}) + cn^d \\ &= cn^d \left(a\left(\frac{1}{b^d}\right)\log n - a\left(\frac{1}{b^d}\right)\log b + 1\right) \\ &= cn^d \left(a\left(\frac{1}{b^d}\right)\log n + 1 - a\left(\frac{1}{b^d}\right)\log b\right). \end{aligned}$$

     Using the fact that $a\left(\frac{1}{b^d}\right)\log b \leq a\left(\frac{1}{b^d}\right)\log n$ for $n \geq b^d$, we can further simplify this inequality to:

$$\begin{aligned} T(n) &\leq cn^d \left(a\left(\frac{1}{b^d}\right)\log n + 1\right) \\ &= O(n^d \log n). \end{aligned}$$

   - Proof of case 2: Suppose $a < b^d$, then we can write $T(n)$ as:

$$T(n) \leq aT\left(\frac{n}{b}\right) + O\left(n^d\right)$$

     Using the recurrence relation for $T(n/b)$, we can write:

$$T(n) \leq a\left(aT\left(\frac{n}{b^2}\right) + O\left(\left(\frac{n}{b}\right)^d\right)\right) + O\left(n^d\right) = a^2 T\left(\frac{n}{b^2}\right) + O\left(n^d\left(\frac{1}{b^d} + 1\right)\right)$$

     Continuing this pattern, we get:

$$T(n) \leq a^j T\left(\frac{n}{b^j}\right) + O\left(n^d \left(\frac{1}{b^d} + \frac{1}{b^{d+1}} + \cdots + \frac{1}{b^{jd}}\right)\right)$$

We can stop when $\frac{n}{b^j} = 1$ or $j = \log_b n$. Then, we get:

$$T(n) \leq a^{\log_b n} T(1) + O\left(n^d \left(\frac{1}{b^d} + \frac{1}{b^{d+1}} + \cdots + \frac{1}{b^{(\log_b n)d}}\right)\right)$$

Since $a < b^d$, we have $a^{\log_b n} < n^d$, so:

$$T(n) = O\left(n^d \left(\frac{1}{b^d} + \frac{1}{b^{d+1}} + \cdots + \frac{1}{b^{(\log_b n)d}}\right)\right)$$

Now, we can use the geometric series formula to simplify the above expression:

$$T(n) = O\left(n^d \frac{1 - \left(\frac{1}{b}\right)^{d \log_b n}}{1 - \frac{1}{b^d}}\right) = O(n^d)$$

Therefore, we have shown that $T(n) = O(n^d)$ when $a < b^d$.

- Proof of case 3: Assume that $a > b^d$. Then we have:

$$T(n) \leq aT\left(\frac{n}{b}\right) + O\left(n^d\right)$$

By the definition of Big-O notation, there exists a constant c $>$ 0 and an integer $n\_0 > 0$ such that:

$$T\left(\frac{n}{b}\right) \leq c\left(\frac{n}{b}\right)^{\log_b a - \epsilon}$$

for all $n \geq n_0$ and some $\epsilon > 0$.

Substituting this into the original equation, we get:

$$T(n) \leq a \cdot c\left(\frac{n}{b}\right)^{\log_b a - \epsilon} + O\left(n^d\right) \quad = c \cdot a^{\log_b n - \epsilon} \cdot n^{\log_b a - \epsilon} + O\left(n^d\right) \; = O\left(n^{\log_b a}\right)$$

where the last step follows from the fact that $a^{\log_b n - \epsilon} = n^{\log_b a - \epsilon}$.

Therefore, we have shown that $T(n) = O\left(n^{\log_b a}\right)$ in Case 3.

Matrix Multiplication

1. Let's write a function using python3 that multiply two matrices A,B (without the use of numpy or any external library).

```
1
2 def matrix_multiply(A, B):
3     #check the dim of each matrix
4     if len(A[0]) != len(B):
5         raise ValueError("Matrix dimensions do not match")
6
7     #define the result matrix
```

```
    result = [[0 for j in range(len(B[0]))] for i in range(len(A))]

    #multiplication
    for i in range(len(A)):
        for j in range(len(B[0])):
            for k in range(len(B)):
                result[i][j] += A[i][k] * B[k][j]

    return result
```

2. The complexity of this algorithm is $O(n^3)$, where n is the number of rows (or columns) of the matrices. This is because each element of the result matrix requires n multiplications and n additions, and there are n^2 elements in total, leading to a total of n^3 operations.

3. To optimize this algorithm we can use **Strassen's algorithm** which is based on divide and conquer approach (like the merge sort algo). Instead of performing 8 multiplications, Strassen's algorithm performs only 7 multiplications.

```
def strassen(A, B):
    n = len(A)
    if n == 1:
        return [[A[0][0] * B[0][0]]]

    #divide matrices A and B into quadrants
    mid = n // 2
    A11 = [row[:mid] for row in A[:mid]]
    A12 = [row[mid:] for row in A[:mid]]
    A21 = [row[:mid] for row in A[mid:]]
    A22 = [row[mid:] for row in A[mid:]]
    B11 = [row[:mid] for row in B[:mid]]
    B12 = [row[mid:] for row in B[:mid]]
    B21 = [row[:mid] for row in B[mid:]]
    B22 = [row[mid:] for row in B[mid:]]

    #calculate the 7 products recursively
    P1 = strassen(add_matrices(A11, A22), add_matrices(B11, B22))
    P2 = strassen(add_matrices(A21, A22), B11)
    P3 = strassen(A11, subtract_matrices(B12, B22))
    P4 = strassen(A22, subtract_matrices(B21, B11))
    P5 = strassen(add_matrices(A11, A12), B22)
    P6 = strassen(subtract_matrices(A21, A11), add_matrices(B11, B12))
    P7 = strassen(subtract_matrices(A12, A22), add_matrices(B21, B22))

    #calculate the result matrix C
    C11 = subtract_matrices(add_matrices(add_matrices(P1, P4), P7), P5)
    C12 = add_matrices(P3, P5)
    C21 = add_matrices(P2, P4)
    C22 = subtract_matrices(add_matrices(subtract_matrices(P1, P2), P3), P6
    )
    C = [[0] * n for _ in range(n)]
    for i in range(mid):
```

```
        for j in range(mid):
            C[i][j] = C11[i][j]
            C[i][j + mid] = C12[i][j]
            C[i + mid][j] = C21[i][j]
            C[i + mid][j + mid] = C22[i][j]

    return C

def add_matrices(A, B):
    n = len(A)
    C = [[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            C[i][j] = A[i][j] + B[i][j]
    return C

def subtract_matrices(A, B):
    n = len(A)
    C = [[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            C[i][j] = A[i][j] - B[i][j]
    return C
```

Quiz

1. What will be the time complexity for the following fragment of code?

```
C=10
B=0
for i in range n:
    B+=i*C
```

Answer: $O(n)$

2. What will be the time complexity for the following fragment of code?

```
i=0
while i<n:
    i*=k
```

Answer: $O(\log_k n)$

3. What will be the time complexity for the following fragment of code?

```
for i in range(n):
    for j in range(m):
```

Answer: $O(n * m)$