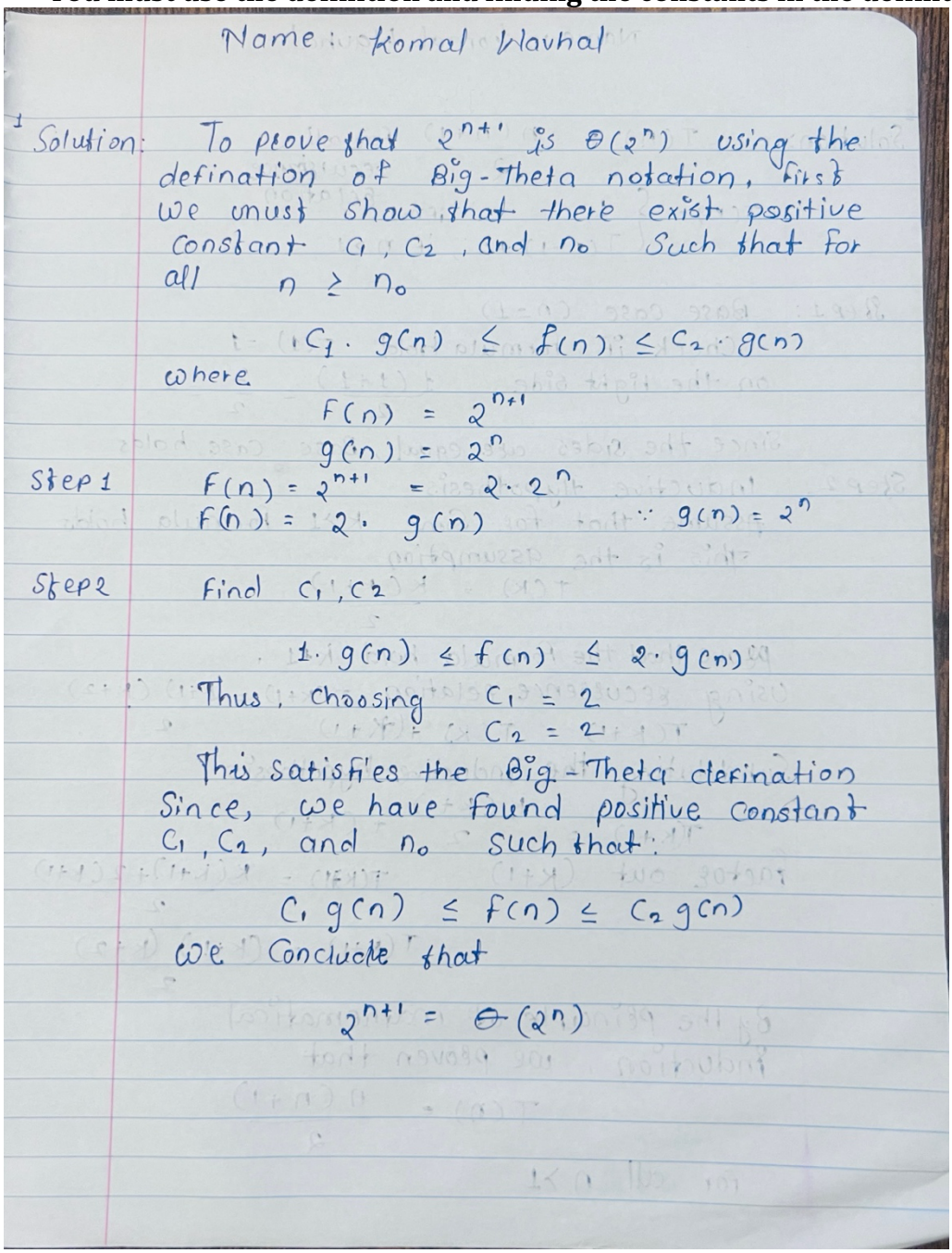
CS600 Midterm Exam Spring 2025

Komal Wavhal

CWID- 20034443

Table of Contents

1. **(6 Points) Using the very definition of Big-Theta notation, prove that 2n+1 is θ(2n). You must use the definition and finding the constants in the definition to receive credit.**



Name : Komal Wavhal

1 Solution: To prove that 2^{n+1} is $\Theta(2^n)$ using the defination of Big-Theta notation, first we must show that there exist positive constant c_1, c_2, and n_0 such that for all $n \geq n_0$

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

where

$$F(n) = 2^{n+1}$$
$$g(n) = 2^n$$

Step 1 $F(n) = 2^{n+1} = 2 \cdot 2^n$

$F(n) = 2 \cdot g(n)$ $\because g(n) = 2^n$

Step 2 Find c_1, c_2

$$1 \cdot g(n) \leq f(n) \leq 2 \cdot g(n)$$

Thus, choosing $c_1 = 2$

$c_2 = 2$

This satisfies the Big-Theta defination Since, we have found positive constant c_1, c_2, and n_0 such that:

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$

We Conclude that

$$2^{n+1} = \Theta(2^n)$$

Big Theta notation is considered a "tight bound" because it precisely characterizes the function's growth rate. When a function's growth rate is within a constant multiplicative factor of another process, we say the function's growth is bound by that factor.

Big Theta notation is used to identify the dominant term of a function as n tends toward infinity. It describes the growth rate of the function as n increases. For example, the function 2n+1 is considered to be 2n because, as n grows large, 2n+1 and 2n grow at the same rate.

Big Theta notation provides a way to express the asymptotic behavior of a function, with $\theta(x)$ indicating its growth rate as x approaches infinity. Since we are primarily concerned with the highest power of a variable in asymptotic analysis, the second term of $2n+1$ becomes insignificant as n approaches infinity. Big Theta notation is commonly used to describe the time complexity of different algorithms.

2. **(7 Points) Given that T(n) = 1 if n=1 and T(n) = T(n-1)+ n otherwise; show, by induction, that T(n) = n(n+1)/2. Show all three steps of your induction explicitly**.

Name: komal Wavhal

2. Solution: $T(n) = \frac{n(n+1)}{2}$ For all recurrence relation

$$T(n) = \begin{cases} 1, & \text{if } n=1 \\ T(n-1)+n, & \text{if } n>1 \end{cases}$$

Step 1: Base case ($n=1$)

Check if formula hold $T(1) = 1$

on the right side $\frac{1(1+1)}{2} = \frac{2}{2} = 1$

Since the sides are equal, Base case holds

Step 2: Inductive Hypothesis

Assume that for some $k \geq 1$ formula holds

this is the assumption

$$T(k) = \frac{k(k+1)}{2}$$

prove that the formula hold $k+1$.

Using recurrence relation $T(k+1) = \frac{(k+1)(k+2)}{2}$

$$T(k+1) = T(k) + (k+1)$$

Substituting the inductive hypothesis

$$T(k+1) = \frac{k(k+1)}{2} + (k+1)$$

factor out $(k+1)$ $T(k+1) = \frac{k(k+1)+2(k+1)}{2}$

$$T(k+1) = \frac{(k+1)(k+2)}{2}$$

By the principle of mathematical induction, we proven that

$$T(n) = \frac{n(n+1)}{2}$$

for all $n \geq 1$

3. **(7 Points) Given the recurrence relation T(n) = 7 T(n/5) + 10n, for n>1; and T(1)=1. Find T(625).**

Name: Komal Wavhal

3 Solution: Given the recurrence relation: $T(n) = 7T(n/5) + 10n$, for $n > 1$
with the base case:
$T(1) : 1$
we need to Compute $T(625)$

Step 1 Expand the recurrence relation
First expansion ($n = 625$):
$T(625) = 7T(125) + 10(625)$
$T(625) = 7T(125) + 6250$

Second Expansion ($n = 125$)
$T(125) = 7T(25) + 10(125)$
$T(125) = 7T(25) + (1250)$

Third Expansion ($n = 25$):
$T(25) = 7T(5) + 10(25)$
$T(25) = 7T(5) + 250$

Fourth Expansion ($n = 5$)
$T(5) = 7T(1) + 10(5)$
Since $T(1) = 1$
$T(5) = 7(1) + 50$
$= 57$

Substitute Back
$T(625)$
1. Compute $T(25)$: $T(25) = 7(57) + 250$
$T(25) = 399 + 250 = 649$
2. Compute $T(125)$:
$T(125) = 7(649) + 1250$
$T(125) = 4543 + 1250 = 5793$
3. Compute $T(625)$:
$T(625) = 7(5793) + 6250$
$T(625) = 40551 + 6250$
$= 46,801$

4. **(16 Points) Suppose that we implement a union-find structure by representing each set using a balanced search tree. Describe and analyze algorithms for each of the methods for a union-find structure so that every operation runs in at most O(logn) time in worst case.**

Name: komal Wavhal

Solution: ① find operation - determine the root of the set containing a perticular element.

Algorithm:
- Start at the node containg the element
- Traverse up the tree by following parent pointer untill the root is found (a node with no parent)
- Return the root of the tree

Time Complexity:

Since the tree is balanced, the height of the tree is $O(\log n)$, where n is the number of elements in the tree.
Therefore, find operation takes $O(\log n)$ time.

Union Operation:

The Union Operation merges two set into one. We can link the root of one tree to the root of the other tree

Algorithm:
- perform find operation on both element to get their roots.
- Compare the heights (or size) of the two tree
- Attach the Shortest tree under the root of the taller tree to maintain balance.

5. **(16 Points). Recall Homework 2 Exercise 2.5.13, where you implemented a stack using two queues.** Now consider implementing a queue using two stacks S1 and S2 where:
   enqueue(o): pushes object o at the top of the stack S1
   dequeue(): if S2 is empty then pop the entire contents of S1 pushing each element onto S2. Then pop from S2.
   If S2 is not empty, pop from S2.

It is easy to see that this algorithm is correct. We are interested in its running time.

a) (4 Points) Show that the conventional worst case running time of a single dequeue is O(n).

b) (12 Points) Show that the amortized cost of a single dequeue is O(1). You must use Amortization Method to receive credit.

5
Solution: Enqueue (o)
operation : push the object o onto Stack S1
Time Complexity : Since pushing an element onto Stack is an O(1) Operation
The enquee (o) operation run in O(1) time

Dequeue ():
Operation : If Stack is empty we move all the element from Stack S1 and Stack S2 by popping each element from S1 and pushing it onto S2.
the reverse order of the element in S1, so the first element inserted into S1 will end up at the top of S2

After this transfer, we pop the top element from Stack S2, which Corresponds to the front element of the queue.

If Stack S2 is not empty, we simply pop the element from S2, which already the front element of the queue.

Time Complexity
in the worst case, when S2 is empty, we need to transfer all n from S1 to S2, which involve n pop and push operation. which take O(n) time

5

once the transfer is complete, popping from S_2 is an $O(1)$ operation.

Time Complexity Analysis

* Worst case time Complexity of Dequeue()
  - when S_2 is empty, dequeue() will transfer all n element from S_1 to S_2
    This involves n pop operation and n push operation onto S_2 taking $O(n)$ time.
  - After the transfer, popping from S_2 is an $O(1)$ operation.
  - Thus, the worst case time Complexity of dequeue() is $O(n)$.

* Amortized Time Complexity of Dequeue()
  The key insight is that each element is moved from S_1 to S_2 at most once and it popped from S_2 only after the transfer. therefore each element can be involved in the transfer and popping operation only once in its lifetime.
  Dividing the total cost by n operations gives an amortized time Complexity of $O(1)$ for each dequeue() operation.

* Enqueue(): $O(1)$
* dequeue():
  worst case $O(n)$ when S_2 empty & need to transfer S_1 to S_2
  Amortized $O(1)$ over multiple dequeue() operation

6. **(16 Points) Consider an n by n matrix M whose elements are 0's and 1's such that in any row, all the 1's come before any 0's in that row**. Assuming A is already in memory, describe an efficient algorithm for finding the row of M that contain the most of 1's. What is the running time of the algorithm?

Solution:
given an n x n matrix M, where each element is either 0 or 1. Additionally, the matrix has the following property: in each row, all the 1's come before any of the 0's.
The goal is to find the row that contains the most number of 1's in the matrix.

Since the 1's in each row are contiguous and all the 1's come before the 0's, we can use this structure to search efficiently. Instead of checking every element in each row, we can use binary search to quickly find the number of 1's in each row.

Binary Search on Each Row:
- For each row, since all 1's are at the beginning, we can use binary search to find the index of the last 1. This will give us the count of 1's in that row directly.

Track the Maximum:
- As we go through each row, we can compare the number of 1's in the row to the current maximum and update the row with the most 1's accordingly.

Algorithm detail steps:
1. Initialize two variables:
   - max_ones = -1 to keep track of the maximum number of 1's found.
   - max_row = -1 to store the index of the row with the most 1's.
2. For each row:
   - Perform a binary search to find the index of the last 1 in that row.
   - The number of 1's in the row is the index of the last 1 + 1.
   - Update the maximum if the current row has more 1's than the previous maximum.
3. Return the row index that has the maximum number of 1's.

Pseudocode:

```
Algorithm FindRow_With_Most_Ones(M):
  Input: M is an n x n matrix where each row contains 0's and 1's
  Output: The index of the row with the most 1's

  max_ones ← -1
  max_row ← -1

  for i ← 0 to n - 1 do:
    low ← 0
    high ← n - 1
    last_one_index ← -1

    # Binary search for the last 1 in row i
    while low ≤ high do:
      mid ← (low + high) / 2
      if M[i][mid] = 1 then:
        last_one_index ← mid
```

```
            low ← mid + 1
        else:
            high ← mid - 1

    # Number of ones in the current row is last_one_index + 1
    num_ones ← last_one_index + 1

    # Update max_ones and max_row if necessary
    if num_ones > max_ones then:
        max_ones ← num_ones
        max_row ← i

  return max_row
```

Input: A matrix A of size n * n, variables row and column which represents the length of rows and columns in the matrix.We are using 0 based indexing

Output: Returning the value of the row with the maximum number of 1's

```
i←0
j←0
row ← 0

while(i < row && j < column) {
        if(A[i][j] == 0) then
                i ← i +1
        else {
                while((j + 1) < column && A[j+ 1] != 0)
                        j++;
                row = i
        }
}
return row;
```

The time complexity of the above algorithm is O(n + n)
which is equivalent to O(n) and the space complexity of the above algorithm is O(1).

1. **Binary Search**:
    - For each row, we perform binary search to find the last occurrence of 1. This is efficient because binary search has a time complexity of **O(log n)**.
    - The number of 1's in that row is last_one_index + 1.
2. **Tracking Maximum**:
    - We update max_ones and max_row whenever we find a row with more 1's than the current maximum.
3. **Overall Complexity**:
    - We are performing **binary search** on each row, and there are **n rows**.
    - The binary search on each row takes **O(log n)** time, so the total time complexity for processing all rows is **O(n log n)**.

Time Complexity:

- **Binary search on each row** takes **O(log n)**.
- Since we process **n** rows, the total time complexity is **O(n log n)**.

Space Complexity:

- The algorithm uses a constant amount of extra space (for variables like max_ones, max_row, low, high, mid, last_one_index), so the space complexity is **O(1)**.

Time complexity: **O(n log n)**, because we perform binary search on each of the **n** rows.
Space complexity: **O(1)**, as we only use a constant amount of extra space.

7. **(16 Points) You are given two sequences A and B of n numbers each, possibly containing duplicates.** Describe an efficient algorithm for determining if A and B contain the same set of numbers, possibly in different orders. What is the running time of this algorithm?

Solution:

To determine if two sequences A and B of size n contain the same set of numbers, possibly in different orders and possibly with duplicates, the goal is to check if A and B contain the same elements with the same frequencies.

Approach:
The most efficient approach is to count the occurrences of each number in both sequences and then compare these counts.
We can do this by leveraging a hash map (or dictionary), which allows for efficient counting of elements.
The algorithm can be broken down into the following steps:

Steps of the Algorithm:

1. Count the occurrences of each element in both sequences:
    - Use two hash maps (or dictionaries), one for sequence A and one for sequence B. Each hash map will store the frequency of each element in the respective sequence.
2. Compare the two hash maps:
    - After building the hash maps for both sequences, simply compare the two hash maps. If they are identical, then A and B contain the same set of numbers with the same frequencies. If they are not identical, then the sequences differ.

Pseudocode:

```
Algorithm AreSame_Set(A, B):
  Input: Two sequences A and B, each of size n
  Output: True if A and B contain the same set of numbers with the same frequencies, False otherwise

  # Create two dictionaries to store the frequencies of elements in A and B
  frequency_A ← empty dictionary
  frequency_B ← empty dictionary

  # Count frequencies of elements in A
  for each element in A do:
    if element is in frequency_A then:
      frequency_A[element] ← frequency_A[element] + 1
    else:
      frequency_A[element] ← 1

  # Count frequencies of elements in B
  for each element in B do:
    if element is in frequency_B then:
      frequency_B[element] ← frequency_B[element] + 1
    else:
      frequency_B[element] ← 1

  # Compare the two frequency dictionaries
  if frequency_A = frequency_B then:
    return True
  else:
```

```
    return False
```

```
def are_same_set(A, B):
    # Step 1: Check if lengths are different (early exit)
    if len(A) != len(B):
        return False

    # Step 2: Create dictionaries to store the frequency counts of elements in A and B
    frequency_A = {}
    frequency_B = {}

    # Step 3: Count frequencies in A
    for element in A:
        if element in frequency_A:
            frequency_A[element] += 1
        else:
            frequency_A[element] = 1

    # Step 4: Count frequencies in B
    for element in B:
        if element in frequency_B:
            frequency_B[element] += 1
        else:
            frequency_B[element] = 1

    # Step 5: Compare the two frequency dictionaries
    if frequency_A == frequency_B:
        return True
    else:
        return False
```

Algorithm: checkIsEqual(int[] A, int[]B)

Input: Two sequences A and B
Output: Returning true if A & B contain same set of numbers

```
Set s1
Set s2
for(i ←0 to A.length-1)
        s1.add(A[i]);
for(i ←0 to A.length-1)
        s2.add(B[i]);

for each element in the set s1
    if( set2 does not contain element) then
        return false;
Return true;
```

Explanation:

1. Counting Frequencies:
    - For each sequence, we iterate over the elements and update their frequency in the respective hash map. The time complexity of inserting or updating an element in a hash map is O(1) on average, so counting the frequencies of all elements in each sequence takes O(n) time.
2. Comparing the Dictionaries:
    - After we have counted the frequencies of elements in both sequences, we compare the two hash maps. Since there are n keys in each hash map, the comparison also takes O(n) time (assuming that comparing two hash maps is done in linear time).

Time Complexity Analysis:

- Counting frequencies for sequence A: O(n)
- Counting frequencies for sequence B: O(n)
- Comparing the two hash maps: O(n)

Thus, the overall time complexity of the algorithm is O(n).

Space Complexity:

- We use two hash maps to store the frequencies of the elements in both sequences. In the worst case, each sequence could have n distinct elements, so the space complexity is O(n) for storing the frequencies.

Time Complexity: O(n), because we iterate over both sequences once to count the frequencies and then perform a linear comparison of the hash maps.

Space Complexity: O(n), due to the storage of the frequency counts in two hash maps.

This approach is efficient and works well for sequences with potentially large sizes, handling both duplicates and different orderings effectively.

8. **(16 Points) Let A and B be two sequences of n integers each, in the range [1, n^4]. Given an integer x, describe an O(n)–time algorithm for determining if there is an integer a in A and an integer b in B such that x=a+b.**

Solution: Utilize a hash set (or dictionary) to store the elements of one of the sequences and then check if the complement (i.e., x - a) exists in the other sequence.
Given two sequences A and B, each of size n, and an integer x, we need to determine if there is an integer a in A and an integer b in B such that x = a + b.

Store elements of sequence B in a hash set:

- First, insert all the elements of sequence **B** into a hash set (or dictionary). This allows for **O(1)** average time complexity for checking if an element exists in **B**.

I**terate through sequence A and check for each element**:

- For each element **a** in sequence **A**, check if **x - a** exists in the hash set of **B**.
- If it exists, then there are integers **a** and **b** such that **x = a + b**, and we can return True.
- If we go through all elements in **A** and do not find a pair, return False.

```
def find_pair_with_sum(A, B, x):
  # Step 1: Create a hash set for sequence B
  set_B = set(B)

  # Step 2: Iterate through elements in A
  for a in A:
    # Check if x - a exists in set_B
    if (x - a) in set_B:
      return True  # Found the pair (a, b) such that x = a + b

  # Step 3: If no such pair is found
  return False
```

Explanation:

1. Building the Hash Set:
   The set(B) operation takes O(n) time to insert n elements from B into the hash set.
2. Iterating through A:
   We then iterate over each element in A and for each element a, we check if x - a is present in the hash set set_B.
   The in set_B check takes O(1) on average due to the hash set's constant-time lookup.
3. Return:
   If we find such a pair, we immediately return True.
   If no such pair is found after checking all elements of A, we return False.

Time Complexity:

- Building the hash set for B: O(n).
- Iterating through A and checking for each element: O(n).

Thus, the total time complexity is O(n).

Space Complexity:

- We use a hash set to store all elements of B, which requires O(n) space.

The algorithm runs in O(n) time and uses O(n) space, making it an efficient solution for the problem of finding two numbers, one from each sequence, that sum to x.

9. **(16 Points). Suppose you are given an instance of the fractional knapsack problem in which all the items have the same weight. Describe an algorithm and provide a pseudo code for this fractional knapsack problem in O(n) time**.

Solution:

In the fractional knapsack problem, you're given a set of items, each with a weight and a value. The goal is to fill a knapsack to maximize the total value, where the knapsack can hold fractions of items, not necessarily whole items.

In this specific case, all items have the same weight, which simplifies the problem significantly. Instead of considering both the value and the weight of each item to determine its efficiency (i.e., value/weight ratio), we only need to focus on the value of the items.

If all items have the same weight, the strategy to maximize the value in the knapsack is straightforward:

1. Sort the items based on their value in descending order.
2. Pick items starting from the highest value, and since all items have the same weight, the order of picking items becomes straightforward.
3. Fill the knapsack as much as possible with whole items, because each item has the same weight.

If the knapsack can hold a fraction of an item, but we are limited to whole items because the weight is fixed, we just pick items from the sorted list until the knapsack is full.

Since all items have the same weight, the optimal strategy is to pick the items with the highest value first, as each item contributes equally to the weight of the knapsack.

Algorithm:

1. **Sort the items** based on their value in descending order.
2. **Add items to the knapsack** in this sorted order. Since all items have the same weight, you can either add a whole item or as many items as fit, depending on the capacity of the knapsack.
3. **Return the total value** of the items added to the knapsack.

```
Algorithm FractionalKnapsackSameWeight(items, capacity):
  Input:
    items: a list of tuples (value, weight) where all items have the same weight
    capacity: the capacity of the knapsack
  Output: the maximum total value that can be obtained in the knapsack

  # Step 1: Sort the items by value in descending order
  Sort items by value in descending order

  # Step 2: Initialize total value and remaining capacity
  total_value ← 0
  remaining_capacity ← capacity
```

```
  # Step 3: Iterate over sorted items and add to knapsack
  for each item (value, weight) in items do:
    if remaining_capacity == 0:
      break  # The knapsack is full

    # Since all items have the same weight, pick the whole item if possible
    if remaining_capacity >= weight:
      total_value ← total_value + value
      remaining_capacity ← remaining_capacity - weight
    else:
      # If the remaining capacity is less than the item weight, take fraction of it
      total_value ← total_value + (value * (remaining_capacity / weight))
      remaining_capacity ← 0

  return total_value
```

```
def fractional_knapsack_same_weight(items, capacity):
  # Step 1: Sort the items by value in descending order
  items.sort(key=lambda x: x[0], reverse=True)  # Sort by value (x[0])

  # Step 2: Initialize total value and remaining capacity of the knapsack
  total_value = 0
  remaining_capacity = capacity

  # Step 3: Iterate over the sorted items and add them to the knapsack
  for value, weight in items:
    if remaining_capacity == 0:
      break  # Knapsack is full, no need to check further

    # If we can fit the entire item into the knapsack
    if remaining_capacity >= weight:
      total_value += value  # Add the whole item's value
      remaining_capacity -= weight  # Decrease the capacity

    else:
      # If we can't fit the whole item, take a fraction of it
      total_value += value * (remaining_capacity / weight)
      remaining_capacity = 0  # Knapsack is full now

  return total_value
```

Explanation:

1. **Sorting**: The algorithm starts by sorting the items based on their **value** in descending order. This ensures that we are considering the most valuable items first.
    - Sorting takes **O(n log n)** time, where **n** is the number of items.
2. **Iterating and Filling the Knapsack**: Once the items are sorted, we iterate through them and add

as many as possible to the knapsack:
    - If the remaining capacity of the knapsack is greater than or equal to the weight of an item, we add it entirely.
    - If the remaining capacity is less than the weight of an item, we take a fraction of that item (since all items have the same weight, this is a simple fraction based on the available capacity).
3. **Stopping Condition**: We stop when the knapsack is full (i.e., when the remaining capacity becomes 0).

Time Complexity:

- **Sorting** the items by value takes **O(n log n)**.
- **Iterating** over the sorted list of items takes **O(n)**.

Thus, the total time complexity of the algorithm is **O(n log n)** due to the sorting step.

Space Complexity:

- The space complexity is **O(n)**, because we store the items in a list and need space for temporary variables like total_value and remaining_capacity.

Even though all items have the same weight, the algorithm still sorts the items based on value and fills the knapsack greedily, which results in an **O(n log n)** time complexity.

Therefore, the algorithm will run in O(n) time complexity because of the following three reasons:
1) To sort n elements, no sorting method is used. As a result, it will not take O (n logn) time.
2) The selection procedure is called iteratively in the complete process and runs in O(n) time.
3) if-else comparisons are applied, these comparisons take O (1) time.

Therefore, the fractional knapsack problem can be solved in O(n) time.

This is the best possible time complexity for this problem due to the sorting step. If the items were to be sorted by another criterion, the problem would be simpler, but the solution's complexity is still dominated by the sorting step.