This program defines a function to compute the Fibonacci numbers using **recursion, and then prints the first 10 Fibonacci numbers (from 0 to 9).**

1. **`recursive_fibonacci(n)` Function**:

This function computes the `n`th Fibonacci number using **recursion**.

```
def recursive_fibonacci(n):
    if n <= 1:
        return n
    else:
        return recursive_fibonacci(n-1) + recursive_fibonacci(n-2)
```

- **Line 1**: `def recursive_fibonacci(n):`

  - This defines a function named `recursive_fibonacci` that takes a single parameter `n` (the position of the Fibonacci number to compute).

- **Line 2**: **`if n <= 1:`**

  - This checks if `n` is either `0` or `1`. These are the **base cases** of the Fibonacci sequence:

    - `F(0) = 0`

    - `F(1) = 1`

- **Line 3**: **`return n`**

  - If `n` is `0` or `1`, the function directly returns `n` (i.e., `F(0) = 0`, `F(1) = 1`).

- **Line 4**: **`else:`**

  - **If `n > 1`**, the function proceeds to the recursive case.

- **Line 5**: **`return recursive_fibonacci(n-1) + recursive_fibonacci(n-2)`**

  - This is the **recursive part**:

    - The `n`th Fibonacci number is the sum of the previous two Fibonacci numbers:

      F(n) = F(n-1) + F(n-2)

    - The function calls itself twice to calculate `F(n-1)` and `F(n-2)` and returns their sum.

Example of `recursive_fibonacci(4)`:

- `F(4)` calls `F(3)` and `F(2)`, which in turn call `F(2)` and `F(1)`, and so on, until it reaches the base cases `F(1)` and `F(0)`.

This recursive method is simple, but **inefficient** because it repeatedly calculates the same Fibonacci values multiple times.

2. **`non_recursive_fibonacci(n)` Function:**

This function computes the Fibonacci sequence up to the `n`th Fibonacci number using an **iterative approach (without recursion).**

```
def non_recursive_fibonacci(n):
    first = 0
    second = 1
    print(first)
    print(second)
    while n - 2 > 0:
        third = first + second
        first = second
        second = third
        print(third)
        n -= 1
```

- **Line 1**: `def non_recursive_fibonacci(n):`
  - This defines a function named `non_recursive_fibonacci` that takes a single parameter `n` (the number of Fibonacci numbers to print).
- **Line 2**: `first = 0`
  - The first Fibonacci number (`F(0)`) is initialized to `0`.
- **Line 3**: `second = 1`
  - The second Fibonacci number (`F(1)`) is initialized to `1`.
- **Line 4**: `print(first)`
  - It prints the first Fibonacci number, `F(0) = 0`.
- **Line 5**: `print(second)`
  - It prints the second Fibonacci number, `F(1) = 1`.
- **Line 6**: `while n - 2 > 0:`
  - This `while` loop runs until we've printed `n` Fibonacci numbers.

- Initially, `n-2 > 0` because we have already printed the first two Fibonacci numbers. The loop continues until `n-2` is no longer greater than 0.

- **Line 7**: `third = first + second`

  - It calculates the next Fibonacci number by adding the two previous Fibonacci numbers (`first` and `second`).

- **Line 8**: `first = second`

  - The value of `first` is updated to be the value of `second` (we shift the Fibonacci sequence).

- **Line 9**: `second = third`

  - The value of `second` is updated to be the value of `third` (we shift the Fibonacci sequence).

- **Line 10**: `print(third)`

  - It prints the next Fibonacci number (`third`), which is the result of adding the previous two numbers.

- **Line 11**: `n -= 1`

  - This decrements `n` by 1 to keep track of how many numbers are left to print. The loop will stop when `n` is no longer greater than 2.

Example of `non_recursive_fibonacci(5)`:

- First, `0` and `1` are printed (the first two Fibonacci numbers).

- Then, the next Fibonacci numbers are calculated iteratively:

  - `F(2) = 1` (printed)

  - `F(3) = 2` (printed)

  - `F(4) = 3` (printed)

3. **Main Block:**

This block of code calls both the recursive and non-recursive functions to print the Fibonacci numbers.

```
if __name__ == "__main__":
    n = 10
    for i in range(n):
        print(recursive_fibonacci(i))

    non_recursive_fibonacci(n)
```

- **Line 1**: `if __name__ == "__main__":`

  - This ensures that the following code runs only if the script is executed directly (not imported as a module).

- **Line 2**: `n = 10`

  - This sets `n = 10`, meaning we want to calculate the first 10 Fibonacci numbers.

- **Line 3**: `for i in range(n):`

  - This `for` loop iterates 10 times (from `i = 0` to `i = 9`).

- **Line 4**: `print(recursive_fibonacci(i))`

  - In each iteration, it calls the `recursive_fibonacci(i)` function to print the `i`th Fibonacci number using recursion.

- **Line 5**: `non_recursive_fibonacci(n)`

  - After printing the first 10 Fibonacci numbers using recursion, it calls the `non_recursive_fibonacci(n)` function to print the first 10 Fibonacci numbers using the non-recursive method.

Conclusion:

- **`recursive_fibonacci(n)`: This function calculates the `n`th Fibonacci number using recursion. It's simple but inefficient for larger values of `n` due to repeated calculations.**

- **`non_recursive_fibonacci(n)`: This function calculates and prints the first `n` Fibonacci numbers using an iterative approach. It's more efficient because it calculates each Fibonacci number only once.**

def fibonacci(i):

```- This line defines a function named `fibonacci` that takes a single argument `i`, which represents the position in the Fibonacci sequence.

if i == 0:

return 0

- This is the **base case** for when `i = 0`. The 0th Fibonacci number is defined to be `0`.

- If `i` is `0`, the function immediately returns `0` without making any further recursive calls.

elif i == 1:

return 1

- This is the **base case** for when `i = 1`. The 1st Fibonacci number is defined to be `1`.

- If `i` is `1`, the function immediately returns `1`.

4. **Recursive Case for i > 1:**

else:

return fibonacci(i-2) + fibonacci(i-1)

- If `i` is greater than 1, the function **recursively calls itself**:

  - `fibonacci(i-2)` calculates the Fibonacci number at position `i-2` (i.e., two positions before `i`).

  - `fibonacci(i-1)` calculates the Fibonacci number at position `i-1` (i.e., one position before `i`).

- The result of these two recursive calls is then **added together**, which gives the Fibonacci number at position `i` because, by definition:

F(i) = F(i-1) + F(i-2)

This is the core formula for the Fibonacci sequence.

for x in range(10):

print(fibonacci(x))

- This loop will run 10 times, with `x` taking values from `0` to `9` (because `range(10)` generates numbers from `0` to `9`).

- For each value of `x`, the function `fibonacci(x)` is called, and the result is printed.

  - For `x = 0`, it calls `fibonacci(0)` which returns `0`.

  - For `x = 1`, it calls `fibonacci(1)` which returns `1`.
```

- For `x = 2`, it calls `fibonacci(2)` which calculates `fibonacci(0) + fibonacci(1)` and returns `1`.

- For `x = 3`, it calls `fibonacci(3)` which calculates `fibonacci(1) + fibonacci(2)` and returns `2`.

- And so on for values from `x = 4` to `x = 9`.

Explanation of Recursion in the Fibonacci Function:

- The recursion allows us to break down the problem of calculating `fibonacci(i)` into smaller subproblems by reducing `i` step by step.

- However, this recursive approach is **inefficient** for larger values of `i` because it recalculates the same Fibonacci numbers multiple times. A more efficient approach would involve **memoization** or **dynamic programming**, but this simple version helps illustrate the basic concept of recursion.

Conclusion:

- The function `fibonacci(i)` uses recursion to compute the Fibonacci number at position `i`.

- It works by breaking the problem down into smaller problems (`fibonacci(i-1)` and `fibonacci(i-2)`).

- The loop prints the Fibonacci numbers for the first 10 values (from `fibonacci(0)` to `fibonacci(9)`), resulting in the sequence starting with `0, 1, 1, 2, 3, 5, 8, 13, 21, 34`.

job_sequencing

def job_sequencing(jobs, n):

  # Sort jobs by profit in descending order

  jobs.sort(key=lambda x: x[2], reverse=True)

```

- **Purpose**: This line sorts the list of jobs based on the profit (which is the 3rd element of each tuple) in descending order.

- **Explanation**: The `lambda x: x[2]` tells Python to use the 3rd element (the profit) of each job tuple for sorting. The `reverse=True` argument sorts the jobs in decreasing order of profit.

  **result = [-1] * n** # To store result of jobs

```

- **Purpose**: Initializes an array `result` with `n` elements, all set to `-1`. This array will hold the job IDs that have been scheduled in each slot.

- **Explanation**: The `result` array represents `n` time slots (one per job). A value of `-1` indicates that a slot is unoccupied. lambda x: x[2] is a **lambda function** that takes an element x from the list (each x represents an item) and returns the **third element** (x[2]) of that item.

In this case, x is assumed to be a list or tuple of the form: [value, weight, value-to-weight ratio].

The lambda function is used to extract the **value-to-weight ratio** (which is the third element in the item) from each element in the list.

Example: If `n = 3`, the `result` array will look like this initially:

result = [-1, -1, -1] # All slots are empty

  total_profit = 0 # Variable to store the total profit

```

- **Purpose**: Initializes a variable `total_profit` to 0. This will keep track of the total profit from the jobs that are successfully scheduled.

  **for job in jobs:**

```

- **Purpose**: Starts a loop that iterates over each job in the `jobs` list (which is sorted by profit).

- **Explanation**: We process each job in the order of highest to lowest profit (because of the earlier sorting).

    for j in range(min(n, job[1]) - 1, -1, -1): # Find a free slot

```

- **Purpose**: This nested loop tries to find an empty slot for the current job.
```

- **Explanation**:- `min(n, job[1]) - 1`: The `job[1]` represents the deadline (the maximum time by which the job should be completed). We use `min(n, job[1])` to ensure that the job doesn't get scheduled beyond the available number of slots (`n`), and subtract `1` because the array index starts from 0.

- `range(min(n, job[1]) - 1, -1, -1)`: This range goes backward from the latest possible slot before the job's deadline (i.e., from `min(n, job[1]) - 1` to 0).

- **Why backward?**: We loop backward to try to assign the job to the latest available slot before its deadline (to maximize the chances of filling the earlier slots with other jobs).

```
        if result[j] == -1:  # If the slot is empty
```

- **Purpose**: This checks if the current slot (`result[j]`) is empty (`-1` means empty).

- **Explanation**: If the slot is empty, it means the job can be scheduled in that slot.

```
            result[j] = job[0]  # Assign job to the slot
```

- **Purpose**: If the slot is empty, this line assigns the current job (i.e., `job[0]`, which is the job ID) to this slot.

- **Explanation**: `job[0]` is the job ID (like 'Job1', 'Job2', etc.), and we place it in the corresponding time slot.

```
            total_profit += job[2]  # Add job profit to total profit
```

- **Purpose**: This line adds the profit of the scheduled job (i.e., `job[2]`, which is the profit of the job) to the total profit.

- **Explanation**: `job[2]` represents the profit of the job, and we're accumulating the profit as jobs are successfully scheduled.

```
            break  # Exit the loop as the job has been scheduled
```

- **Purpose**: This breaks out of the inner loop because we've successfully scheduled the job in a slot.

- **Explanation**: Once the job is assigned to a slot, we don't need to check other slots for this job, so we exit the inner loop.

---

return [r for r in result if r != -1], total_profit

```

- **Purpose**: Returns two things:

  1. A list of jobs that have been scheduled (i.e., jobs that are not `-1`).

  2. The total profit from all scheduled jobs.

- **Explanation**: The list comprehension `[r for r in result if r != -1]` filters out the empty slots (i.e., `-1` values) and returns only the scheduled jobs (those that have a job ID).

  - For example, if `result = ['Job1', 'Job3', -1]`, the returned list will be `['Job1', 'Job3']`.

**### Example Usage:**

**jobs = [('Job1', 2, 100), ('Job2', 1, 19), ('Job3', 2, 27), ('Job4', 1, 25), ('Job5', 3, 15)]**

**scheduled_jobs, max_profit = job_sequencing(jobs, 3)**

**print(f"Scheduled jobs: {scheduled_jobs}")**

**print(f"Total profit: {max_profit}")**

```

- **Explanation: Here, the jobs are provided as tuples where each tuple contains:**

  - `job_id` (e.g., 'Job1')

  - `deadline` (the last time the job can be completed)

  - `profit` (the profit earned by completing the job)

- **Expected Output:**

  ```python

  Scheduled jobs: ['Job3', 'Job1']

  Total profit: 127

Summary:

- The function first sorts jobs based on profit, then iterates through each job and tries to find the latest available slot for it before its deadline.

- If a slot is available, the job is scheduled, and its profit is added to the total profit

This code solves the **Fractional Knapsack Problem using the **greedy approach**. Let's break down the code line by line to understand how it works.**

Code Explanation:

arr = [[60, 10], [100, 20], [120, 30]]

w = 50

price = 0

```

- `arr`: This is a list of items, where each item is represented by a list with two elements:

  - The first element is the **value** of the item.

  - The second element is the **weight** of the item.

  - So, the list `arr` represents the following items:

    - Item 1: Value = 60, Weight = 10

    - Item 2: Value = 100, Weight = 20

    - Item 3: Value = 120, Weight = 30

- `w`: This is the total capacity of the knapsack, which is 50 in this case. It represents the maximum weight the knapsack can carry.

- `price`: This variable stores the total value of the items that we are putting into the knapsack. We initialize it to `0`.

**---**

**arr = sorted(arr, key=lambda x: x[0] / x[1], reverse=True)**

```

- **Sorting**: This line sorts the items in `arr` based on their **value-to-weight ratio** (`value / weight`).

  - `lambda x: x[0] / x[1]` calculates the value-to-weight ratio for each item. Here `x[0]` is the value, and `x[1]` is the weight.

  - `reverse=True` sorts the items in descending order, meaning that the item with the highest value-to-weight ratio comes first.

- **After sorting**:

  - Item 1: Value = 60, Weight = 10, Ratio = 60/10 = 6.0

  - Item 2: Value = 100, Weight = 20, Ratio = 100/20 = 5.0

  - Item 3: Value = 120, Weight = 30, Ratio = 120/30 = 4.0

- Sorted `arr` will look like this:

  arr = [[60, 10], [100, 20], [120, 30]]

for i in range(len(arr)):

```

- **Loop through all items**: This loop iterates over each item in the sorted `arr` list (which is now sorted by value-to-weight ratio).

**itemWt = arr[i][1]**

**itemP = arr[i][0]**

```

- For each item, we extract:

  - `itemWt`: The weight of the current item (`arr[i][1]`).

  - `itemP`: The value of the current item (`arr[i][0]`).

if(itemWt > w):

price += w * (itemP / itemWt)

break

```

- **If the item can't fit completely**: If the weight of the current item (`itemWt`) is greater than the remaining capacity of the knapsack (`w`), we can only take a **fraction** of the item.

  - `itemP / itemWt`: This gives the value-to-weight ratio of the item.

  - `w * (itemP / itemWt)`: We take the fraction of the item's value based on the remaining capacity (`w`).

  - `break`: Once we have taken the fraction of an item, the knapsack is full (`w = 0`), so we stop the loop with `break`.

**else:**

**price += itemP**

**w -= itemWt**

```

- **If the item fits entirely**: If the entire item can fit in the knapsack (i.e., `itemWt <= w`):

- We add the full value of the item (`itemP`) to `price`.
- We decrease the remaining capacity of the knapsack by the weight of the item (`w -= itemWt`).

print(price)

```

- **Print the result**: This line prints the total value (`price`) of the items in the knapsack.

**### Example Walkthrough:**

**Given the input:**

**```python**

**arr = [[60, 10], [100, 20], [120, 30]]**

**w = 50**

```

1. **Sort items by value-to-weight ratio:**

After sorting, the items are in the following order:

- Item 1: Value = 60, Weight = 10, Ratio = 6.0

- Item 2: Value = 100, Weight = 20, Ratio = 5.0

- Item 3: Value = 120, Weight = 30, Ratio = 4.0

2. **Start filling the knapsack:**

- **Item 1 (Value = 60, Weight = 10):**

- The knapsack has enough space (50 units), so we take the whole item.

- `price += 60` (total value becomes 60).

- Remaining capacity: `50 - 10 = 40`.

- **Item 2 (Value = 100, Weight = 20):**

- The knapsack still has enough space (40 units), so we take the whole item.

- `price += 100` (total value becomes 160).

- Remaining capacity: `40 - 20 = 20`.

- **Item 3** (Value = 120, Weight = 30):
  - The knapsack only has 20 units of remaining capacity, which is less than the item's weight (30).
  - We take a fraction of the item: `price += 20 * (120 / 30) = 20 * 4 = 80`.
  - `price` becomes 160 + 80 = 240.
  - Now, the knapsack is full (`w = 0`), so we stop.

3. **Result**:
  - The total value in the knapsack is `240`.

Final Output:

```
240
```

Time Complexity:

- **Sorting the items**: O(n log n), where `n` is the number of items.
- **Looping through items**: O(n), where `n` is the number of items.

Thus, the overall time complexity is **O(n log n)**, dominated by the sorting step.

Conclusion:

This code effectively solves the fractional knapsack problem by greedily taking the most valuable items first, and if needed, taking fractions of items to maximize the total value in the knapsack.

This code solves the **0/1 Knapsack Problem using **recursive dynamic programming**.**

Initialization of Variables

def solve_knapsack():

val = [50, 100, 150, 200] # Value array: The value of each item

wt = [8, 16, 32, 40] # Weight array: The weight of each item

W = 64 # The total weight capacity of the knapsack

n = len(val) - 1 # n represents the index of the last item, i.e., 3 in this case (for 4 items)

```

- `val`: A list of values for each item.

- Item 1 has a value of 50, Item 2 has a value of 100, Item 3 has a value of 150, and Item 4 has a value of 200.

- `wt`: A list of weights for each item.

- Item 1 has a weight of 8, Item 2 has a weight of 16, Item 3 has a weight of 32, and Item 4 has a weight of 40.

- `W`: The total weight capacity of the knapsack (64 in this case).

- `n`: The number of items minus 1 (i.e., `n = 3` since there are 4 items, and we use 0-based indexing).

**### 2. **Defining the Knapsack Recursive Function**:**

**def knapsack(W, n):** # W = remaining weight capacity, n = number of items considered so far

```

- The function `knapsack(W, n)` is a **recursive function** that calculates the maximum value that can be obtained for a given weight capacity `W` using the first `n + 1` items.

3. **Base Case:**

if n < 0 or W <= 0:

return 0

- This is the **base case** for the recursion.

- If there are no items left to consider (`n < 0`) or if the knapsack has no remaining capacity (`W <= 0`), the maximum value is `0` because no further items can be added to the knapsack.

4. **Checking If the Item Can Fit**:

```
if wt[n] > W:
    return knapsack(W, n-1)
```

- If the weight of the current item `wt[n]` is greater than the remaining capacity of the knapsack `W`, this item cannot be added to the knapsack.

- In that case, the function **recursively calls itself** but with one less item (`n-1`), essentially excluding this item from the selection.

5. **Decision to Include or Exclude the Current Item**:

```
else:
    return max(val[n] + knapsack(W-wt[n], n-1), knapsack(W, n-1))
```

- If the current item can fit in the knapsack (`wt[n] <= W`), the function has two choices:

1. **Include the current item**: Add its value `val[n]` to the result of the recursive call with reduced weight capacity (`W-wt[n]`), and decrease the number of items considered (`n-1`).

2. **Exclude the current item**: Simply move to the next item without including the current one, i.e., call `knapsack(W, n-1)`.

- The function returns the maximum of these two options (i.e., either including the item or excluding it). This ensures that the **best value** is chosen.

6. **Final Output**:

```
print(knapsack(W, n))
```

- The final answer is obtained by calling the `knapsack(W, n)` function with the full weight capacity `W` and considering all the items (from `n=3` to `n=0`).

- This will compute the maximum value that can be achieved within the given weight capacity.

Example Walkthrough:

Given the input:

- `val = [50, 100, 150, 200]` (Values of the 4 items).

- `wt = [8, 16, 32, 40]` (Weights of the 4 items).

- `W = 64` (Knapsack capacity).

We need to find the maximum value that can be carried in the knapsack.

First Recursion:

- We start with `W = 64` and all 4 items.

- The function first checks if the 4th item (value 200, weight 40) can fit into the knapsack:

  - Yes, because 40 <= 64.

  - We now have two options:

    - **Include item 4**: Add 200, reduce the capacity to 64 - 40 = 24, and solve the problem for the remaining items with capacity 24.

    - **Exclude item 4**: Solve the problem with the remaining capacity 64 and consider only the first 3 items.

Continue Recursively:

- This process continues, each time making a decision whether to include the current item or exclude it. The recursion works through all items and all possible combinations of items, considering their values and weights.

Final Result:

- After all the recursive calls, the function computes the maximum value that can be achieved with the given capacity.

Output:

When the function is called with `W = 64` and `n = 3` (considering all items), the output will be:

350

```

This means the maximum value that can be packed in the knapsack with a capacity of 64 units is 350.

### Time Complexity:

- **Time complexity**: The time complexity of this recursive solution is **O(2^n)**, where `n` is the number of items. This is because the function explores all possible combinations of items by making two recursive calls for each item (include or exclude). The exponential complexity makes this approach inefficient for larger input sizes.

### Conclusion:

This code uses a recursive approach to solve the **0/1 knapsack problem**, which considers each item either included or excluded, based on whether it fits in the knapsack. The solution is optimal, but it can be computationally expensive for larger problems due to the exponential growth of the recursive calls.
```

The code implements two variants of the **Quick Sort algorithm:**

1. **Deterministic Quick Sort: This variant picks the **middle element** as the pivot.**

2. **Randomized Quick Sort: This variant picks a **random element** from the array as the pivot.**

Code Explanation:

import random

```

- **`import random`**: This imports the `random` module, which is used to randomly select a pivot in the **Randomized Quick Sort**.

**#### Deterministic Quick Sort**

**def quick_sort_deterministic(arr):**

```

- **`def quick_sort_deterministic(arr):`**: This defines the function `quick_sort_deterministic`, which takes an array `arr` as input and returns a sorted version of the array using the deterministic version of the quicksort algorithm.

```
 if len(arr) <= 1:
     return arr
```

- **Base Case**: If the length of `arr` is 1 or 0, the array is already sorted. So, we return the array as is.

```
  pivot = arr[len(arr) // 2]
```

- **Choosing the Pivot**: The pivot is selected as the **middle element** of the array. We find the middle index using `len(arr) // 2` (integer division).

```
  left = [x for x in arr if x < pivot]
```

- **Left Subarray**: We create a list `left` that contains all the elements from the array that are smaller than the pivot. This is done using a **list comprehension**: `[x for x in arr if x < pivot]`.

middle = [x for x in arr if x == pivot]

```

- **Middle Subarray**: We create a list `middle` that contains all elements equal to the pivot. This accounts for cases where there are duplicate pivot values in the array. Again, we use list comprehension: `[x for x in arr if x == pivot]`.

**right = [x for x in arr if x > pivot]**

```

- **Right Subarray**: We create a list `right` that contains all the elements from the array that are greater than the pivot: `[x for x in arr if x > pivot]`.

return quick_sort_deterministic(left) + middle + quick_sort_deterministic(right)

```

- **Recursive Step**: We recursively sort the `left` and `right` subarrays and then concatenate them with the `middle` subarray. The `middle` subarray contains all elements equal to the pivot, so it is already "sorted". The final result is the concatenation of the sorted left subarray, the middle elements (pivot), and the sorted right subarray.

**---**

**#### Randomized Quick Sort**

**def quick_sort_randomized(arr):**

```

- **`def quick_sort_randomized(arr):`**: This defines the function `quick_sort_randomized`, which implements the randomized quicksort algorithm.

if len(arr) <= 1:

return arr

```

- **Base Case**: If the length of `arr` is 1 or 0, the array is already sorted, so we return the array as is.

**pivot_index = random.randint(0, len(arr) - 1)**

```

- **Random Pivot Selection**: We randomly select a pivot index using the `random.randint()` function, which generates a random integer between `0` and `len(arr) - 1`. This ensures the pivot is chosen randomly from the array.

pivot = arr[pivot_index]

- **Pivot Value**: Once the pivot index is randomly selected, we get the pivot value from the array using `arr[pivot_index]`.

left = [x for x in arr[:pivot_index] + arr[pivot_index + 1:] if x < pivot]

```

- **Left Subarray**: We create the `left` subarray using list comprehension. This includes all elements from the array that are smaller than the pivot:

- We combine two parts of the array (excluding the pivot): `arr[:pivot_index]` (elements before the pivot) and `arr[pivot_index + 1:]` (elements after the pivot).

- We then filter out elements that are smaller than the pivot using `if x < pivot`.

**middle = [pivot]**

```

- **Middle Subarray**: The pivot is placed in the middle subarray. Since the pivot is the element selected randomly, there is only one instance of it in the middle list.

right = [x for x in arr[:pivot_index] + arr[pivot_index + 1:] if x > pivot]

```

- **Right Subarray**: Similarly, we create the `right` subarray, which contains all elements greater than the pivot. The same two parts of the array are combined (excluding the pivot), and then we filter out elements that are greater than the pivot using `if x > pivot`.

**return quick_sort_randomized(left) + middle + quick_sort_randomized(right)**

```

- **Recursive Step**: Like in the deterministic quick sort, we recursively sort the `left` and `right` subarrays and then concatenate them with the `middle` subarray**.**

Example Usage

arr = [3, 7, 8, 5, 2, 1, 9, 6, 4]

```

- This is the initial array of integers that we want to sort using both the deterministic and randomized quicksort algorithms.

**print("Deterministic Quick Sort:")**

**print(quick_sort_deterministic(arr.copy()))**

```

- We call the `quick_sort_deterministic` function to sort the array and print the result. We use `arr.copy()` to avoid modifying the original array when calling the deterministic quick sort.

print("Randomized Quick Sort:")

print(quick_sort_randomized(arr.copy()))

```

### Sample Output:

```

Deterministic Quick Sort:

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

Randomized Quick Sort:

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

```

**---**

**### Key Points:**

**1. **Deterministic Quick Sort** always uses the middle element as the pivot.**

**2. **Randomized Quick Sort** selects a random pivot from the array to help avoid worst-case scenarios (e.g., when the pivot is always the smallest or largest element).**

**3. Both algorithms use the same basic structure of quicksort: partitioning the array into left, middle, and right subarrays, then recursively sorting the left and right subarrays.**

**4. The code uses **list comprehensions** to create the left, middle, and right subarrays in a concise and readable manner.**
```

def n_queens(n):

```

- **`def n_queens(n):`**: This defines the main function `n_queens`, which will solve the N-Queens problem for a given board size `n`.

```
    col = set()
    posDiag = set() # (r+c) for diagonals
    negDiag = set() # (r-c) for diagonals
```

- **`col`**: This is a set used to keep track of which columns are already occupied by a queen. If `col` contains an index `c`, it means the queen is already placed in column `c`.

- **`posDiag`**: This is a set used to track the positive diagonals. The positive diagonal is characterized by the sum of the row and column indices `(r + c)`. If `(r + c)` is in `posDiag`, a queen is already placed on that diagonal.

- **`negDiag`**: This is a set for the negative diagonals. The negative diagonal is characterized by the difference `(r - c)`. If `(r - c)` is in `negDiag`, a queen is already placed on that diagonal.

```
    res = []
```

- **`res`**: This list will store all the valid solutions found by the backtracking algorithm. Each solution will be represented as a list of strings, where each string represents a row of the chessboard.

```
    board = [["0"] * n for i in range(n)]
```

- **`board`**: This initializes the board as an `n × n` grid filled with `"0"`s. `"0"` means there is no queen in that position. The board is represented as a list of lists, where each inner list corresponds to a row on the board.

```
    def backtrack(r):
```

- **`def backtrack(r):`**: This is the recursive helper function used for backtracking. It tries to place queens row by row.

  - **`r`** represents the current row we're trying to place a queen in.
```

```
        if r == n:
            copy = [" ".join(row) for row in board]
            res.append(copy)
            return
```

- **Base case**: If `r == n`, it means we've successfully placed queens in all rows (i.e., the board is completely filled with queens), so we have found a valid solution.

- **`copy = [" ".join(row) for row in board]`**: We create a copy of the current board and format each row as a string of space-separated values (e.g., `"1 0 0 0"`). This ensures the board is printed clearly.

- **`res.append(copy)`**: The valid solution is added to the `res` list.

- **`return`**: Once a solution is found, we return from the recursive function to explore other possibilities.

```
        for c in range(n):
            if c in col or (r + c) in posDiag or (r - c) in negDiag:
                continue
```

- **Loop over each column**: We loop through all columns `c` to try and place a queen in the current row `r`.

- **`if c in col or (r + c) in posDiag or (r - c) in negDiag:`**: This condition checks if placing a queen at position `(r, c)` is safe:
  - If column `c` is already occupied (`c in col`).
  - If the positive diagonal `(r + c)` is already occupied.
  - If the negative diagonal `(r - c)` is already occupied.

- **`continue`**: If any of these conditions is true, we skip this column and try the next one.

```
            col.add(c)
            posDiag.add(r + c)
            negDiag.add(r - c)
            board[r][c] = "1"
```

- **Place a queen**: If it’s safe to place a queen at `(r, c)`, we:

- Add `c` to the `col` set to mark that column as occupied.

- Add `(r + c)` to the `posDiag` set to mark that diagonal as occupied.

- Add `(r - c)` to the `negDiag` set to mark the other diagonal as occupied.

- Place the queen on the board by setting `board[r][c] = "1"`, where `"1"` represents a queen.

```
        backtrack(r + 1)
```

- **Recursive call**: After placing a queen, we recursively call the `backtrack` function to place queens in the next row (`r + 1`).

```
        col.remove(c)
        posDiag.remove(r + c)
        negDiag.remove(r - c)
        board[r][c] = "0"
```

- **Backtracking**: If placing the queen in the next row leads to a conflict, we "undo" the placement (i.e., backtrack):

- Remove `c` from `col`, `(r + c)` from `posDiag`, and `(r - c)` from `negDiag` to mark those columns and diagonals as unoccupied.

- Set `board[r][c] = "0"` to remove the queen from the current position.

```
    backtrack(0)
```

- **Start backtracking**: We start the backtracking process by calling `backtrack(0)`, which tries to place queens starting from the first row (`r = 0`).

```
    for sol in res:
        for row in sol:
            print(row)
        print()
```

- **Print the solutions**: Once all solutions have been found:

- We loop over each solution in `res`.

- For each solution, we print each row (which is a string of space-separated values).

- An empty line is printed between different solutions for clarity.

```
if __name__ == "__main__":
    n_queens(4)
```

- **Main execution**: This calls the `n_queens` function with `n = 4` to solve the 4-Queens problem.

Sample Output:

For `n = 4`, the possible solutions would be:

```
1 0 0 0
0 0 1 0
0 1 0 0
0 0 0 1

0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
```

Explanation of the Output:

- The two solutions show how the queens are placed on the 4x4 chessboard.

- `1` represents a queen, and `0` represents an empty space.

- The queens in each solution are placed such that they don't threaten each other (i.e., no two queens are in the same row, column, or diagonal).

Key Concepts:

- **Backtracking**: The algorithm places queens one by one in each row and backtracks when it encounters an invalid configuration.

- **Sets for optimization**: We use sets (`col`, `posDiag`, `negDiag`) to efficiently track which columns and diagonals are occupied by a queen, making it quick to check if a placement is safe.

Time Complexity:

- The time complexity of this backtracking solution is **O(N!)**, where `N` is the size of the chessboard (`n × n`). This is because for each row, we attempt to place a queen in each of the `n` columns, and in the worst case, we try all possible configurations.

Explanation of the Code for Solving TSP (Travelling Salesman Problem) Using Branch and Bound:

This code uses **Branch and Bound** to solve the **Travelling Salesman Problem (TSP)**. Let's go through it step by step.

1. Class Initialization (`__init__` method)

```python
class TSPSolver:
    def __init__(self, graph):
        self.n = len(graph)        # The number of cities (nodes) is the length of the graph.
        self.graph = graph         # The adjacency matrix representing the distances between cities.
        self.final_path = [None] * (self.n + 1)  # To store the best path found.
        self.visited = [False] * self.n  # To track which cities have been visited.
        self.final_res = math.inf  # Variable to store the minimum cost (initialized to infinity).
```

- `graph` is an adjacency matrix where `graph[i][j]` represents the distance between city `i` and city `j`.
- `final_path` is the array that will store the best path (i.e., the sequence of cities).
- `visited` is a boolean list that keeps track of whether a city has been visited in the current path.
- `final_res` stores the minimum cost of completing the TSP, initially set to infinity.

2. `first_min` Method

```python
def first_min(self, i):
    """Returns the minimum edge weight (not equal to i) from node i."""
    return min([self.graph[i][k] for k in range(self.n) if i != k])
```

- `first_min(i)` calculates the smallest weight (or cost) for a node `i` in the graph. It does this by checking all other nodes `k` where `i != k`, and returns the minimum distance between node `i` and any other node.

3. `second_min` Method

```python
def second_min(self, i):
    """Returns the second minimum edge weight (not equal to i) from node i."""
    return sorted([self.graph[i][k] for k in range(self.n) if i != k])[:2]
```

- `second_min(i)` calculates the second smallest edge weight for node `i`. It sorts the distances to other nodes and returns the two smallest values.

4. `tsp_recursive` Method

```python
def tsp_recursive(self, curr_bound, curr_weight, level, curr_path):
    """Recursive function to solve TSP using Branch and Bound."""
    if level == self.n:
        # If we've visited all cities, check if the current path is valid and calculate the result
        if self.graph[curr_path[level - 1]][curr_path[0]] != 0:
            curr_res = curr_weight + self.graph[curr_path[level - 1]][curr_path[0]]
            if curr_res < self.final_res:
                self.final_res = curr_res
                self.final_path[:] = curr_path[:] + [curr_path[0]]
        return
```

- This is the main recursive method that explores possible paths in the TSP problem.

- **Base Case:** If all cities have been visited (i.e., `level == self.n`), the function checks if the current path is valid (i.e., the return trip to the starting city is possible, i.e., `graph[curr_path[level - 1]][curr_path[0]] != 0`). If so, it calculates the total cost (`curr_res`), and if this is smaller than the previous best (`final_res`), it updates the best cost and stores the current path.

```python
    for i in range(self.n):
        if self.graph[curr_path[level - 1]][i] and not self.visited[i]:
            temp_bound = curr_bound
            curr_weight += self.graph[curr_path[level - 1]][i]

            # Update the bound if we are at the first or other levels
            if level == 1:
                curr_bound -= (self.first_min(curr_path[level - 1]) + self.first_min(i)) / 2
            else:
                curr_bound -= (self.second_min(curr_path[level - 1])[1] + self.first_min(i)) / 2

            # If the bound plus current weight is less than the best result found, explore further
            if curr_bound + curr_weight < self.final_res:
                curr_path[level] = i
                self.visited[i] = True
                self.tsp_recursive(curr_bound, curr_weight, level + 1, curr_path)

            # Backtrack: unmark the current city as visited
            curr_weight -= self.graph[curr_path[level - 1]][i]
            curr_bound = temp_bound
            self.visited[i] = False  # Correctly unmark city i as visited
```

- **Exploration:** For each city `i` that is not yet visited and has a valid edge (`self.graph[curr_path[level - 1]][i]`), we calculate the potential new bound for the solution and add the current edge weight (`curr_weight`).

- If the current bound plus the accumulated weight (`curr_weight`) is less than the best cost (`final_res`), we recursively explore this path by marking city `i` as visited (`self.visited[i] = True`) and move to the next level.

- **Backtracking:** After the recursive call, we unmark the current city as visited (`self.visited[i] = False`), remove the edge weight from `curr_weight`, and restore the bound to its previous state (`curr_bound = temp_bound`).

5. `solve` Method

```python
def solve(self):
    """Starts solving the TSP problem."""
    curr_bound = math.ceil(sum(self.first_min(i) + self.second_min(i)[1] for i in range(self.n)) / 2)
    curr_path = [-1] * (self.n + 1)
    self.visited[0] = True
    curr_path[0] = 0
    self.tsp_recursive(curr_bound, 0, 1, curr_path)
```

- **Initial Setup:** We start solving the TSP by calculating an initial bound (`curr_bound`). This bound is computed by summing the first minimum and second minimum edge weights for each city, divided by 2, and then rounding it up to the nearest integer (`math.ceil`).

- **Starting Point:** The first city (city 0) is marked as visited (`self.visited[0] = True`), and the path starts from city 0 (`curr_path[0] = 0`).

- The `tsp_recursive` method is called to start the recursive search with the initial bound, zero weight, and the initial path.

6. `print_solution` Method

```python
def print_solution(self):
    """Prints the solution."""
    print("Minimum cost:", self.final_res)
    print("Path:", ' -> '.join(map(str, self.final_path)))
```

```

- This method prints the minimum cost (`final_res`) and the optimal path (`final_path`) that solves the TSP problem.

### **7. Example Usage**

```python
graph = [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]]
solver = TSPSolver(graph)
solver.solve()
solver.print_solution()
```

- `graph` is the adjacency matrix representing the distances between 4 cities. For example, `graph[0][1] = 10` means the distance between city 0 and city 1 is 10 units.
- The `TSPSolver` class is instantiated with this graph.
- The `solve()` method is called to compute the optimal path and minimum cost.
- The `print_solution()` method is called to print the results.

Final Output Example

```plaintext
Minimum cost: 80
Path: 0 -> 1 -> 3 -> 2 -> 0
```

- The minimum cost of completing the TSP tour is 80 units.
- The optimal path that visits all cities exactly once and returns to the starting city is: `0 -> 1 -> 3 -> 2 -> 0`.

Summary
```

- This code implements the **Travelling Salesman Problem (TSP)** using the **Branch and Bound** technique.

- The `first_min` and `second_min` methods help in bounding the search space.

- The `tsp_recursive` method explores possible paths using recursion while pruning invalid or suboptimal paths based on the current bound.

- The `solve` method initializes the search, and `print_solution` outputs the result once the optimal path and cost have been found.

Let's go through this code step-by-step and explain each part in simple terms. This code implements the **Merge Sort** algorithm using **multithreading** to speed up the sorting process.

1. Importing the `threading` module

```python
import threading
```

- This imports the `threading` module, which allows the program to create multiple threads. Threads are used to execute different parts of the code concurrently, helping to potentially speed up the sorting process by performing operations in parallel.

---

2. The `merge` function

```python
def merge(arr, left, mid, right):
    L, R = arr[left:mid + 1], arr[mid + 1:right + 1]
    i = j = 0
    for k in range(left, right + 1):
        if j >= len(R) or (i < len(L) and L[i] <= R[j]):
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
```

- **Purpose:** This function **merges two sorted halves** of the array into a single sorted array.
- **Parameters:**

- `arr`: The array to be sorted.

- `left`, `mid`, `right`: Indices that define the boundaries of the two halves to be merged.

- **How it works:**

- The array is divided into two sub-arrays, `L` (left part) and `R` (right part).

- `L` contains the elements from `arr[left]` to `arr[mid]`, and `R` contains the elements from `arr[mid+1]` to `arr[right]`.

- The `for` loop goes through the indices from `left` to `right`, comparing the elements in `L` and `R` and placing the smaller element into the original array `arr`.

- The idea is to maintain two pointers (`i` and `j`) for `L` and `R`, respectively, and put the smaller element between `L[i]` and `R[j]` into the array `arr[k]`.

---

3. The `merge_sort` function

```python
def merge_sort(arr, left, right):
    if left < right:
        mid = (left + right) // 2
        merge_sort(arr, left, mid)
        merge_sort(arr, mid + 1, right)
        merge(arr, left, mid, right)
```

- **Purpose:** This is the main **recursive merge sort function** that divides the array into smaller sub-arrays and sorts them.

- **Parameters:**

- `arr`: The array to be sorted.

- `left` and `right`: The starting and ending indices of the sub-array to be sorted.

- **How it works:**

- The `merge_sort` function repeatedly splits the array into two halves until each sub-array contains only one element (which is trivially sorted).

- The array is split by calculating the `mid` index and recursively calling `merge_sort` on the left half (`arr[left, mid]`) and the right half (`arr[mid+1, right]`).

- After the recursion reaches the base case (when `left == right`), it calls the `merge` function to combine the two sorted halves back into a single sorted array.

---

4. The `threaded_merge_sort` function

```python
def threaded_merge_sort(arr, left, right, depth=2):
    if left < right:
        mid = (left + right) // 2
        if depth > 0:
            left_thread = threading.Thread(target=threaded_merge_sort, args=(arr, left, mid, depth - 1))
            right_thread = threading.Thread(target=threaded_merge_sort, args=(arr, mid + 1, right, depth - 1))
            left_thread.start()
            right_thread.start()
            left_thread.join()
            right_thread.join()
        else:
            merge_sort(arr, left, mid)
            merge_sort(arr, mid + 1, right)
        merge(arr, left, mid, right)
```

- **Purpose:** This function is a **multithreaded version** of the merge sort algorithm. It uses threads to recursively sort both halves of the array in parallel (up to a specified depth).
- **Parameters:**
  - `arr`: The array to be sorted.
  - `left`, `right`: Indices defining the current sub-array being sorted.

- `depth`: This parameter determines how deep the threads should go. When `depth` reaches 0, it switches back to using normal (non-threaded) `merge_sort`.

- **How it works:**

  - If the depth is greater than 0, the function creates two threads: one to sort the left half (`left_thread`) and one to sort the right half (`right_thread`).

  - `left_thread.start()` and `right_thread.start()` start these threads to run the sorting of the left and right sub-arrays in parallel.

  - After both threads are started, `left_thread.join()` and `right_thread.join()` ensure that the main program waits for both threads to complete before proceeding.

  - If `depth` is 0, instead of using threads, it calls the regular `merge_sort` function to sort the halves without creating new threads.

  - After sorting both halves, the `merge` function is called to merge the two sorted halves back into the original array.

---

5. Main Program

```python
arr = list(map(int, input("Enter numbers to sort, separated by spaces: ").split()))
threaded_merge_sort(arr, 0, len(arr) - 1)
print("Sorted array:", arr)
```

- **Purpose:** This part handles user input and calls the `threaded_merge_sort` function to sort the array.

- **How it works:**

  - `input("Enter numbers to sort, separated by spaces: ").split()` reads a line of numbers from the user, splits them into individual strings, and then `map(int, ...)` converts those strings into integers.

  - `arr = list(...)` stores the list of integers.

  - `threaded_merge_sort(arr, 0, len(arr) - 1)` sorts the entire array `arr` from index `0` to `len(arr) - 1` using the threaded version of merge sort.

  - Finally, the sorted array is printed.

How This Code Works Overall:

1. **User Input:** The user is prompted to input a list of numbers separated by spaces.

2. **Sorting with Threads:** The `threaded_merge_sort` function is called, which will attempt to sort the list using threads if the `depth` is greater than 0. Each thread recursively splits and sorts two halves of the array.

3. **Merging Sorted Halves:** Once the array is divided enough (either using threads or regular recursion), the `merge` function merges the sorted halves back into the original array.

4. **Final Output:** After the entire array is sorted, the sorted array is printed.

Summary of Key Points:

- **Merge Sort:** A divide-and-conquer algorithm that recursively divides the array into halves, sorts each half, and then merges them.

- **Multithreading:** The program uses threads to sort the left and right halves of the array concurrently, potentially speeding up the sorting process.

- **Depth Control:** The `depth` parameter controls how deep the recursive threads should go before switching to normal recursion, which helps prevent creating too many threads.