Name: Suraj Balaso Desai

Worked Examples

Problem Statement 1

In DNA sequence analysis, it is crucial to identify similarities and differences between different sequences in order to understand genetic variation and develop treatments for genetic diseases. However, with the increasing amount of genetic data being generated, it becomes challenging to efficiently compare and analyze these sequences. Therefore, there is a need for an efficient sorting algorithm that can handle large datasets of DNA sequences and accurately identify similarities and differences between them.

Solution :

The examination of DNA sequences is a critical component of genetic research, which offers valuable insights into the genes' functions and structures. However, DNA sequencing generates vast amounts of data that need efficient and precise analysis to extract relevant information. Comparing two or more DNA sequences involves arranging and aligning them to identify similarities or differences, which can be difficult and time-consuming. Therefore, developing effective algorithms to analyze DNA sequences is essential to enhance our comprehension of genetic material and its role in biological systems. By using merge sort to sort and compare DNA sequences, the analysis's speed and accuracy can be significantly improved, making it a crucial problem to address in the field of bioinformatics.

Here's an example list of DNA sequences that can be sorted using Merge sort:

1. AGCTAGCTAGCTAGCT
2. GCTAGCTAGCTAGCTA
3. CTAGCTAGCTAGCTAG
4. TAGCTAGCTAGCTAGC
5. AGCTAGCTAGCTAGCT
6. CGTAGCTAGCTAGCTA
7. TAGCTAGCTAGCTAGC
8. GCTAGCTAGCTAGCTA
9. TAGCTAGCTAGCTAGC
10. CGTAGCTAGCTAGCTA

Note: This is just an example and not a real DNA sequence.

If we sort the DNA sequences using merge sort, we can compare them more efficiently to identify similarities or differences.
For example, we could identify common sequences or mutations that may be linked to certain diseases or traits.
By using merge sort, we can efficiently sort and compare large datasets of DNA sequences, which is critical in bioinformatics research.

A general approach to solve this problem is:

1. Input the DNA sequences that require comparison.

2. Divide the DNA sequences into smaller sub-sequences of the same length.
   The length of the sub-sequences can differ depending on the application and input sequence size.
3. Use merge sort to sort the sub-sequences of each DNA sequence, creating a set of sorted sub-sequences for each input sequence.
4. Compare the sorted sub-sequences from each DNA sequence to find similarities or differences.
   This can be accomplished by matching or contrasting each pair of sub-sequences from various input sequences.
5. Reassemble the original DNA sequence, highlighting the similarities or differences found by merging the sorted sub-sequences back together.

Steps to solve the problem

First, we need to obtain a dataset of DNA sequences that we want to compare.
Let's assume we have the following three DNA sequences:

Sequence 1: ATCGTACG
Sequence 2: TCGTAGCA
Sequence 3: GCTAGTAC

Next, we can implement the merge sort algorithm to sort these sequences in lexicographical order.

Here's the step-by-step process for merging two sorted sequences:

1. Create a temporary array to hold the merged sequence.
2. Compare the first element of each sequence.
3. Take the smaller of the two elements and add it to the temporary array.
4. Move to the next element in the sequence from which the smaller element was taken.
5. Repeat steps 2-4 until all elements in one of the sequences have been added to the temporary array.
6. Add the remaining elements from the other sequence to the temporary array.
7. The temporary array now contains the merged sequence.

We can recursively apply this merging process to sort all three sequences. Here's the process:

1. Divide the sequences into smaller sub-sequences until each sub-sequence contains only one element.
2. Merge the sub-sequences in pairs to create sorted sequences that contain two elements each.
3. Repeat step 2 until all sequences have been merged into a single, sorted sequence.

After applying the merge sort algorithm, we can obtain the following sorted sequences:

Sorted Sequence 1: ACGCGTTT
Sorted Sequence 2: ACGGTTCT
Sorted Sequence 3: ACGGTTAT

Finally, we can compare these sorted sequences to identify similarities or differences between them. For example, we can see that the first six characters of Sequence 2 and Sequence 3 are identical, whereas Sequence 1 differs from the other two sequences at the 3rd index.

That's how we can use merge sort to sort DNA sequences and compare them in bioinformatics.

Once it is sorted then we can:

1. Finding common substrings: Once the sequences are sorted, we can scan them to identify common substrings. This information can be useful in identifying regions of similarity between different sequences.

2. Identifying mutations: We can compare the sorted sequences to a reference sequence to identify mutations or differences. This can be important in understanding the evolution of a particular organism or in diagnosing genetic disorders.

3. Clustering sequences: By comparing the sequences, we can group them into clusters based on their similarity. This can be useful in identifying subtypes or strains of a virus, for example.

Code

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]
    left = merge_sort(left)
    right = merge_sort(right)
    return merge(left, right)

def merge(left, right):
    result = []
    i = 0
    j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result += left[i:]
    result += right[j:]
    return result

# Example DNA sequences
sequences = ["AGTCCTAGTGGATAGCCATG", "AGTCCTAGTGGACAGCCATG",
"AGTCCTAGTGGATGGCCATG"]

sorted_sequences = []

# Sort the sequences using merge sort
for sequence in sequences:
    sorted_sequences.append(merge_sort(sequence))

# Compare the sorted sequences.
# Now her below we can write comparison logic based on use case
# Right now I am just comparing if they are equal
```

```
matching_indices = set()
non_matching_indices = []

for i in range(len(sorted_sequences)):
    for j in range(i+1, len(sorted_sequences)):
        if sorted_sequences[i] == sorted_sequences[j]:
            matching_indices.append((i, j))
        else:
            non_matching_indices.append((i, j))

matching_indices_str = ", ".join([f"{i}, {j}" for (i, j) in
matching_indices])
matching_indices_str = ", ".join([f"{i}, {j}" for (i, j) in
non_matching_indices])
print("Matching indices:", matching_indices_str)
print("Non-matching indices:", matching_indices_str)
```

Output

```
Matching indices: 0, 2

Non-matching indices: 1
```

Problem Statement 2

In large investment fund, and we need to constantly monitor and analyze the performance of thousands of stocks in order to make informed investment decisions. We want to maintain a sorted view of the stock prices, sorted by their closing prices, but we also need to constantly update this view as new stock prices become available throughout the trading day.

Solution :

Efficiently maintaining a sorted view of a large and constantly updating data set is a common problem in many real-world applications. For example, financial institutions often need to process large volumes of stock price data in real-time to make informed trading decisions. In order to maintain a sorted view of this data, they need an efficient algorithm that can merge and insert new data into the existing sorted set without performing a full sort operation each time. This not only saves computational resources but also ensures that the sorted view is constantly up-to-date and accurate. Therefore, solving this problem can have significant practical implications for a wide range of industries and applications.

Steps to solve the problem

Here is a general approach to solving the problem of maintaining a sorted view of a constantly updated and expanding dataset without performing a full sort operation each time:

1. Begin with an initial set of data that is already sorted (if possible), or sort the entire dataset using a fast and efficient sorting algorithm such as merge sort.
2. As new data is added to the dataset, use a combination of insertion sort and merge sort to maintain a sorted view of the data.
3. Whenever new data is added, insert it into the sorted set using an insertion sort algorithm. This ensures that the new data is placed in its correct sorted position.
4. Once a certain threshold of new data has been reached (e.g., a certain number of new elements or a certain percentage increase in the size of the dataset), use a merge sort algorithm to merge the new data with the existing sorted set.
5. Repeat steps 3 and 4 as new data is continuously added to the dataset, maintaining a sorted view of the data at all times.
6. By using this approach, we can efficiently maintain a sorted view of a constantly updated and expanding dataset without incurring the high computational costs associated with performing a full sort operation each time.

For example, suppose we receive the following orders:

Order 1: $30
Order 2: $50
Order 3: $40

We start by inserting the first order into the sorted list:

Sorted list: [Order 1]

Next, we insert the second order by merging it with the existing list:

Sorted list: [Order 1, Order 2]

Finally, we insert the third order by performing an insertion sort operation on the merged list:

Sorted list: [Order 1, Order 3, Order 2]

By using merge-insertion sort, we can efficiently maintain a sorted view of the orders as they arrive in real-time, without the need for a full sort operation each time.

Code

Function:

```
def merge_insertion_sort(arr):
    # Divide the array into subarrays of size 32
    subarrays = [arr[i:i+32] for i in range(0, len(arr), 32)]

    # Sort each subarray using insertion sort
    for i in range(len(subarrays)):
        insertion_sort(subarrays[i])

    # Repeatedly merge pairs of subarrays until there is only one subarray
left
    while len(subarrays) > 1:
        merged = []
        for i in range(0, len(subarrays), 2):
            if i == len(subarrays) - 1:
                merged.append(subarrays[i])
            else:
                merged.append(merge(subarrays[i], subarrays[i+1]))
        subarrays = merged

    # The final subarray is the sorted array
    return subarrays[0]


def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j+1] = arr[j]
            j -= 1
```

```
        arr[j+1] = key


def merge(arr1, arr2):
    result = []
    i = j = 0
    while i < len(arr1) and j < len(arr2):
        if arr1[i] <= arr2[j]:
            result.append(arr1[i])
            i += 1
        else:
            result.append(arr2[j])
            j += 1
    result.extend(arr1[i:])
    result.extend(arr2[j:])
    return result
```

Driver Code:

```
stock_prices = [105.4, 107.2, 103.1, 110.8, 111.2, 109.7, 112.1, 113.5,
115.2, 111.9, 116.3, 118.7, 119.2, 120.1, 121.3]

# Sort the data using merge-insertion sort
sorted_data = merge_insertion_sort(stock_prices)

# Print the sorted data
print(sorted_data)
```

Output:

```
[103.1, 105.4, 107.2, 109.7, 110.8, 111.2, 111.9, 112.1, 113.5, 115.2,
116.3, 118.7, 119.2, 120.1, 121.3]
```

In conclusion, maintaining a sorted view of large sets of data that are constantly being updated or modified is a challenging problem. However, by using a combination of merge and insertion sort algorithms, we can efficiently merge new data into an existing sorted set without the need to perform a full sort operation each time. This approach allows for real-time processing of new data and efficient maintenance of a sorted view of the data as it changes over time. In the case of the stock prices example, this approach can be used to quickly sort and process new orders as they arrive, allowing for timely and accurate trading decisions.