Replacement Selection

Declaration

I hereby declare that all the work done in this project titled "*Replacement Selection*" is of my independent effort.

Chapter 1: Introduction

When the input is much too large to fit into memory, we have to do **external sorting** instead of internal sorting. One of the key steps in external sorting is to generate sets of sorted records (also called **runs**) with limited internal memory. The simplest method is to read as many records as possible into the memory, and sort them internally, then write the resulting run back to some tape. The size of each run is the same as the capacity of the internal memory.

Replacement Selection sorting algorithm was described by Donald Knuth. Notice that as soon as the first record is written to an output tape, the memory it used becomes available for another record. Assume that we are sorting in ascending order, if the next record is not smaller than the record we have just output, then it can be included in the run. My job is to implement this replacement selection algorithm.

Chapter 2: Algorithm Specification

Data Structure:

`status[M]` : Used to record the status of data in memory, where 0 indicates participation in the current round of sorting and can be output in this round; 1 indicates participation in the next round of sorting and can only be output in the next round.

`output[N]` : Stores data that can be output in the current round, facilitating the output process.

`array[M]` : Memory used for internal sorting, serving as the storage location for input data.

Algorithm:

The task can be sloved in the following way:

1. Fill the memory, setting the status of all data to 0.
2. Find the minimum value among the data with status 0, store it in the output array for future output, and then read in new data to replace this minimum value.
3. If the newly entered data is smaller than the minimum value from the previous round, set its status to 1, awaiting output in the next round. Otherwise, repeat step 2.

4. When all data in the memory have a status of 1, reset all statuses to 0, reset the output array, and initiate a new sorting round.

Chapter 3: Testing Results

Sample Input:

```
13 3
81 94 11 96 12 99 17 35 28 58 41 75 15
```

Sample Output:

```
11 81 94 96 99
12 17 28 35 41 58 75
15
```

Chapter 4: Analysis and Comments

Time Complexity Analysis:

1. **Initialization Phase:**
    - Initializing the `status` and `output` arrays takes O(M+N) time.
2. **Input Phase:**
    - Reading M elements into the `array` array has a time complexity of O(M).
3. **Main Loop Phase:**
    - The outer loop runs (N-M) times, and the inner loop runs M times.
    - In the inner loop, calling the `FindMin` function, which, in the worst case, needs to traverse the entire `status` array, has a time complexity of O(M).
    - Therefore, the time complexity of the main loop is O((N-M) * M).
4. **Final Output Phase:**
    - The last call to the `Print` function, which needs to traverse the `output` array, has a time complexity of O(N).

Overall Time Complexity:

- O(M+N) + O(M) + O((N-M) * M) + O(N) = O(N * M)

Space Complexity Analysis:

- The usage of three arrays (`status` , `output` , and `array`) contributes to a space complexity of O(N).

In summary, the time complexity of the code is O(N * M), and the space complexity is O(N).