Question 05:

(i)

```
#include <iostream>
using namespace std;

// Function to heapify a subtree rooted at index `i` for a subarray A[st] to A[ed]
void heapify(int A[], int st, int ed, int i) {
   int smallest = i;         // Initialize smallest as root
   int left = 2 * (i - st) + 1 + st;  // Calculate left child index
   int right = 2 * (i - st) + 2 + st; // Calculate right child index

   // Check if left child exists and is smaller than root
   if (left <= ed && A[left] < A[smallest]) {
      smallest = left;
   }

   // Check if right child exists and is smaller than the current smallest
   if (right <= ed && A[right] < A[smallest]) {
      smallest = right;
   }

   // If smallest is not root, swap and continue heapifying
   if (smallest != i) {
      swap(A[i], A[smallest]);
      heapify(A, st, ed, smallest);
   }
}
```

```
// Function to sort a subarray A[st] to A[ed] using heap sort
void heapSortSubarray(int A[], int st, int ed) {
   int n = ed - st + 1; // Size of the subarray

   // Step 1: Build a min-heap for the subarray
   for (int i = st + n / 2 - 1; i >= st; --i) {
      heapify(A, st, ed, i);
   }

   // Step 2: Extract elements from the heap and sort
   for (int i = ed; i > st; --i) {
      // Move smallest element (root) to the end of the subarray
      swap(A[st], A[i]);

      // Reduce the heap size and heapify the root
      heapify(A, st, i - 1, st);
   }
}

// Utility function to print the array
void printArray(int A[], int N) {
   for (int i = 0; i < N; ++i) {
      cout << A[i] << " ";
   }
   cout << endl;
}

int main() {
   // Example array and indices
```

```
    int A[] = {10, 3, 7, 5, 2, 8, 4};
    int N = sizeof(A) / sizeof(A[0]);
    int st = 2, ed = 5;

    cout << "Original array: ";
    printArray(A, N);

    // Sort the subarray
    heapSortSubarray(A, st, ed);

    cout << "Array after sorting subarray: ";
    printArray(A, N);

    return 0;
}
```

(ii)

Brute Force:

Text T: 000000010100000010

Pattern P: 010

Brute Force Algorithm:

1. Compare P with T[1-3]: 000 (mismatch)
2. Compare P with T[2-4]: 000 (mismatch)
3. Compare P with T[3-5]: 000 (mismatch)
4. Compare P with T[4-6]: 000 (mismatch)
5. Compare P with T[5-7]: 000 (mismatch)

6. Compare P with T[6-8]: 010 (match)

7. Compare P with T[7-9]: 100 (mismatch)

8. Compare P with T[8-10]: 000 (mismatch)

9. Compare P with T[9-11]: 000 (mismatch)

10. Compare P with T[10-12]: 010 (match)

11. Compare P with T[11-13]: 000 (mismatch)

Number of comparisons: 11

KMP:

Text T: 0 0 0 0 0 0 1 0 1 0 0 0 0 1 0

Pattern P: 0 1 0

i (text index): 0

j (pattern index): 0

Compare T[i] and P[j]. If they match, increment both i and j. If they don't match, update j using the lps table.

Comparisons:

i j T[i] P[j] Action

0 0 0 0 i++, j++

1 1 0 1 j = lps[j] = 0

2 0 0 0 i++, j++

...

6 0 1 0 j = lps[j] = 0

7 0 0 0 i++, j++

8 1 1 1 i++, j++

9 2 0 0 i++, j++ (pattern found)

...

i j T[i] P[j]

10 0 0   0   i++, j++

Number of comparisons: 15