Heap

Lesson-3

Merge k sorted arrays

Given k sorted arrays of size n each, merge them and print the sorted output.

Input:

k = 3, n = 4

arr[][] = { {1, 3, 5, 7},

{2, 4, 6, 8},

{0, 9, 10, 11}} ;

Output: *0 1 2 3 4 5 6 7 8 9 10 11*

Explanation: *The output array is a sorted array that contains all the elements of the input matrix.*

Approach:

1. Create a min Heap and insert the first element of all k arrays.
2. Run a loop until the size of MinHeap is greater than zero.
3. Remove the top element of the MinHeap and print the element.
4. Now insert the next element from the same array in which the removed element belonged.
5. If the array doesn't have any more elements, then replace root with infinite.After replacing the root, heapify the tree.

```
int *mergeKArrays(int arr[][n], int k)
{

```

```
// To store output array
    int *output = new int[n*k];

    // Create a min heap with k heap nodes.
    // Every heap node has first element of an array
    MinHeapNode *harr = new MinHeapNode[k];
    for (int i = 0; i < k; i++)
    {

// Store the first element
        harr[i].element = arr[i][0];

// index of array
        harr[i].i = i;

// Index of next element to be stored from the array
        harr[i].j = 1;
    }

// Create the heap
    MinHeap hp(harr, k);

    // Now one by one get the minimum element from min
    // heap and replace it with next element of its array
    for (int count = 0; count < n*k; count++)
    {
        // Get the minimum element and store it in output
        MinHeapNode root = hp.getMin();
        output[count] = root.element;

        // Find the next elelement that will replace current
        // root of heap. The next element belongs to same
        // array as the current root.
        if (root.j < n)
        {
            root.element = arr[root.i][root.j];
            root.j += 1;
        }
```

```
            // If root was the last element of its array
    // INT_MAX is for infinite
    else root.element =  INT_MAX;

            // Replace root with next element of array
            hp.replaceMin(root);
        }

        return output;
    }
```

Complexity Analysis:

- **Time Complexity :**O(n * k * log k), Insertion and deletion in a Min Heap requires log k time. So the Overall time complexity is O(n * k * log k)
- **Space Complexity :**O(k), If Output is not stored then the only space required is the Min-Heap of k elements. So space Complexity is O(k).

Height of a complete binary tree (or Heap) with N nodes

Let the size of heap be **N** and height be **h**

If we take a few examples, we can notice that the value of h in a complete binary tree is ceil($\log_2(N+1)$) – 1.

Examples :

N	h
1	0
2	1
3	1
4	2
5	2

```
int main() {
int height(int N)
{
    return ceil(log2(N + 1)) - 1;
}
}
```

Largest Derangement of a Sequence

Given any sequence S=a1+a2+a3….., find the largest derangement of S.
A derangement D is any permutation of S , such that no two elements at the same position in D and S are equal.
The Largest Derangement is such that Di > Di+1.

Examples:

Input : seq[] = {5, 4, 3, 2, 1}
Output : 4 5 2 1 3

Approach:
Since we are interested in generating the largest derangement, we start putting larger elements in more significant positions.
Start from the left, at any **i** position place the next largest element among the values of the sequence which have not yet been placed in positions before **i** .
To scan all positions takes N iteration. In each iteration we are required to find a maximum numbers, so a trivial implementation would be O($n2$) complexity,
However if we use a data structure like max-heap to find the maximum element, then the complexity reduces to O($n * logn$)

```
void printLargest(int seq[], int N)
{
    int res[N]; // Stores result

    // Insert all elements into a priority queue
    std::priority_queue<int> pq;
    for (int i = 0; i < N; i++)
        pq.push(seq[i]);

    // Fill Up res[] from left to right
    for (int i = 0; i < N; i++) {
        int d = pq.top();
        pq.pop();
        if (d != seq[i] || i == N - 1) {
            res[i] = d;
        } else {

            // New Element poped equals the element
            // in original sequence. Get the next
            // largest element
            res[i] = pq.top();
            pq.pop();
            pq.push(d);
        }
    }

    // If given sequence is in descending order then
    // we need to swap last two elements again
    if (res[N - 1] == seq[N - 1]) {
        res[N - 1] = res[N - 2];
        res[N - 2] = seq[N - 1];
    }
```

Time Complexity:O(n log n)

Smallest Derangement of a Sequence

Given the sequence

S=1,2,3....N

find the lexicographically smallest (earliest in dictionary order) derangement of S.

A derangement of S is as any permutation of S such that no two elements in S and its permutation occur at the same position.

Input: 3
Output : 2 3 1
Explanation: The Sequence is 1 2 3.
Possible permutations are (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2) (3, 2, 1).
Derangements are (2, 3, 1), (3, 1, 2).
Smallest Derangement: (2, 3, 1)

APPROACH :

Using a min heap we can successively get the least element and place them in more significant positions, taking care that the property of derangement is maintained.

```
void generate_derangement(int N)
{
  // Generate Sequence and insert into a
   // priority queue.
   int S[N + 1];
```

```
    priority_queue<int, vector<int>, greater<int> > PQ;
    for (int i = 1; i <= N; i++) {
        S[i] = i;
        PQ.push(S[i]);
    }

    // Generate Least Derangement
    int D[N + 1];
    for (int i = 1; i <= N; i++) {
        int d = PQ.top();
        PQ.pop();
        if (d != S[i] || i == N) {
            D[i] = d;
        }
        else {
            D[i] = PQ.top();
            PQ.pop();
            PQ.push(d);
        }
    }

    if (D[N] == S[N])
       swap(D[N-1], D[N]);

    // Print Derangement
    for (int i = 1; i <= N; i++)
        printf("%d ", D[i]);
    printf("\n");
}
```

Complexity: O(N * log N)

K maximum sum combinations from two arrays

Given two equally sized arrays (A, B) and N (size of both arrays).

A **sum combination** is made by adding one element from array A and another element of array B. Display the **maximum K valid sum combinations** from all the possible sum combinations.

Examples:

```
Input :  A[] : {3, 2}
        B[] : {1, 4}
        K : 2 [Number of maximum sum
            combinations to be printed]
Output : 7    // (A : 3) + (B : 4)
        6    // (A : 2) + (B : 4)
```

Approach 1 (Naive Algorithm) :
We can use Brute force through all the possible combinations that can be made by taking one element from array A and another from array B and inserting them to a max heap. In a max heap maximum element is at the root node so whenever we pop from max heap we get the maximum element present in the heap. After inserting all the sum combinations we take out K elements from max heap and display it.

```
 void KMaxCombinations(int A[], int B[],
                       int N, int K)
{
  // max heap.
  priority_queue<int> pq;

  // insert all the possible combinations
```

```
        // in max heap.
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                pq.push(A[i] + B[j]);

        // pop first N elements from max heap
        // and display them.
        int count = 0;
        while (count < K) {
            cout << pq.top() << endl;
            pq.pop();
            count++;
        }
    }
```

Time Complexity: O(N2)

Approach 2 (Sorting, Max heap, Map) :

Instead of brute-forcing through all the possible sum combinations, we should find a way to limit our search space to possible candidate sum combinations.

1. Sort both arrays array A and array B.
2. Create a max heap i.e priority_queue in C++ to store the sum combinations along with the indices of elements from both arrays A and B which make up the sum. Heap is ordered by the sum.
3. Initialize the heap with the maximum possible sum combination i.e (A[N – 1] + B[N – 1] where N is the size of array) and with the indices of elements from both arrays (N – 1, N – 1). The tuple inside the max heap will be (A[N-1] + B[N – 1], N – 1, N – 1). Heap is ordered by first value i.e sum of both elements.
4. Pop the heap to get the current largest sum and along with the indices of the element that make up the sum. Let the tuple be (sum, i, j).

1. Next insert (A[i – 1] + B[j], i – 1, j) and (A[i] + B[j – 1], i, j – 1) into the max heap but make sure that the pair of indices i.e (i – 1, j) and (i, j – 1) are not already present in the max heap. To check this we can use set in C++.
2. Go back to 4 until K times.

```cpp
void KMaxCombinations(vector<int>& A,
                      vector<int>& B, int K)
{
   // sort both arrays A and B
   sort(A.begin(), A.end());
   sort(B.begin(), B.end());

   int N = A.size();

   // Max heap which contains tuple of the format
   // (sum, (i, j)) i and j are the indices
   // of the elements from array A
   // and array B which make up the sum.
   priority_queue<pair<int, pair<int, int> > > pq;

   // my_set is used to store the indices of
   // the  pair(i, j) we use my_set to make sure
   // the indices doe not repeat inside max heap.
   set<pair<int, int> > my_set;

   // initialize the heap with the maximum sum
   // combination ie (A[N - 1] + B[N - 1])
   // and also push indices (N - 1, N - 1) along
   // with sum.
   pq.push(make_pair(A[N - 1] + B[N - 1],
                     make_pair(N - 1, N - 1)));

   my_set.insert(make_pair(N - 1, N - 1));

```

```cpp
    // iterate upto K
    for (int count = 0; count < K; count++)
    {
        // tuple format (sum, (i, j)).
        pair<int, pair<int, int> > temp = pq.top();
        pq.pop();

        cout << temp.first << endl;

        int i = temp.second.first;
        int j = temp.second.second;

        int sum = A[i - 1] + B[j];

        // insert (A[i - 1] + B[j], (i - 1, j))
        // into max heap.
        pair<int, int> temp1 = make_pair(i - 1, j);

        // insert only if the pair (i - 1, j) is
        // not already present inside the map i.e.
        // no repeating pair should be present inside
        // the heap.
        if (my_set.find(temp1) == my_set.end())
        {
            pq.push(make_pair(sum, temp1));
            my_set.insert(temp1);
        }

        // insert (A[i] + B[j - 1], (i, j - 1))
        // into max heap.
        sum = A[i] + B[j - 1];
        temp1 = make_pair(i, j - 1);

        // insert only if the pair (i, j - 1)
        // is not present inside the heap.
        if (my_set.find(temp1) == my_set.end())
        {
            pq.push(make_pair(sum, temp1));
            my_set.insert(temp1);
        }
    }
}
```

Time Complexity : O(N log N) assuming K <= N