Algorithm Challenge solution by Norma Dani Risdiandita

The Naive solution is given below (saved as totalCost_Naive.py)

```
def totalCost(scores):
    answer = 10*len(scores)
    for i in range(len(scores)):
        for j in range(0,i):
            if scores[i] > scores[j]:
                answer += 1
    return answer

scores = [250, 1820, 870, 1000, 2000]

print(totalCost(scores))

scores = [1675, 4660, 7028, 8022, 2529, 3270, 2472, 420, 3024, 5501,
4647, 313, 3568, 4105, 7372, 8680, 1966, 3952, 5320, 7663,
2828, 5868, 286, 9149, 7979, 6050, 1070, 5388]

print(totalCost(scores))
```

```
58
496
```

The quicker solution is using Fenwick tree (credit to: Raju Varshney who wrote the code on https://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/). I tried to use Raju's Fenwick tree code and my implementation is shown below (saved as totalCost_Not_Naive.py)

```
#=====================================================
# The function to perform Binary Indexed Tree is
# imported from
# https://www.geeksforgeeks.org/binary-indexed-tree-or-fenwick-tree-2/
# Special Thanks to Raju Varshney who wrote the code
#=====================================================

# Returns sum of arr[0..index]. This function assumes
# that the array is preprocessed and partial sums of
# array elements are stored in BITree[].
def getsum(BITTree,i):
    s = 0 #initialize result
```

```
    # index in BITree[] is 1 more than the index in arr[]
    i = i+1

    # Traverse ancestors of BITree[index]
    while i > 0:

        # Add current element of BITree to sum
        s += BITTree[i]

        # Move index to parent node in getSum View
        i -= i & (-i)
    return s

# Updates a node in Binary Index Tree (BITree) at given index
# in BITree. The given value 'val' is added to BITree[i] and
# all of its ancestors in tree.
def updatebit(BITTree , n , i ,v):

    # index in BITree[] is 1 more than the index in arr[]
    i += 1

    # Traverse all ancestors and add 'val'
    while i <= n:

        # Add 'val' to current node of BI Tree
        BITTree[i] += v

        # Update index to that of parent in update View
        i += i & (-i)


# Constructs and returns a Binary Indexed Tree for given
# array of size n.
def construct(arr, n):

    # Create and initialize BITree[] as 0
    BITTree = [0]*(n+1)

    # Store the actual values in BITree[] using update()
    for i in range(n):
        updatebit(BITTree, n, i, arr[i])
```

```
    # Uncomment below lines to see contents of BITree[]
    #for i in range(1,n+1):
    #     print BITTree[i],
    return BITTree

# here below my written code

def totalCost(scores):
    # initialize an array of zeros to allocate the number
    # of occurence items. From the problem, scores[i]
    # have values between 0 and 10000. For safety reason,
    # I would choose 12000
    occurence = [0] * 12000

    # initialize the reward
    answer = 0
    BITTree = construct(occurence, len(occurence))
    for val in scores:
        occurence[val] += 1
        # updating the BITTree
        updatebit(BITTree, len(occurence), val, 1)
        # adding the reward and the bonus
        answer += 10 + getsum(BITTree, val - 1)
    return answer

# initialize the inputs, the scores1, scores2, and scores3
# and scoressfinal are the sample test cases

scores1 = [250, 1820, 870, 1000, 2000]
scores2 = [1874, 1339, 5617, 8331, 5424, 9667]
scores3 = [1675, 4660, 7028, 8022, 2529, 3270, 2472, 420, 3024, 5501,
4647, 313, 3568, 4105, 7372, 8680, 1966, 3952, 5320, 7663,
2828, 5868, 286, 9149, 7979, 6050, 1070, 5388]
scoresfinal = list(range(1,11112))*9

# printing the results

print("totalCost 1 = {}".format(totalCost(scores1)))
print("totalCost 2 = {}".format(totalCost(scores2)))
print("totalCost 3 = {}".format(totalCost(scores3)))

# the printing of scoresfinal is wrapped in the time checker
```

```
# to determine the computation time

import time
start = time.time()
print("totalCost 4 = {}".format(totalCost(scoresfinal)))
end = time.time()
print("the time to execute the scoresfinal is {0:.2f} seconds".format(end -
start))
```

The resulting output is given by

```
totalCost 1 = 58
totalCost 2 = 72
totalCost 3 = 496
totalCost 4 = 2778472215
the time to execute the scoresfinal is 0.36 seconds
```