Caleb Howard
S5155126
2801ICT

Maximize The Score Solver C++

Overview

Maximize the Score is a problem in which there are *n* values and two players. Each player gets a maximum of *k* turns to choose a value, after *k* turns the next player has their turns. One of the players will always choose the highest value, whilst the other will choose the value with the highest sum of digits.
This document will include an explanation of the solver, a description of how it works, results, a correctness analysis and a performance analysis.

Solution

The solver uses a single custom priority queue class to store the values according to each players' priority. The following is a detailed description of each component in the class.

mNodes and mSumNodes are two vectors that store pairs of integers, which we will call nodes. For each node (pair) the first item is a value that a player can choose, the second item is the sum of digits of that value. The mNodes vector is sorted in a max heap configuration based on the first item in each pair (value), whereas the mSumNodes vector is sorted in a max heap configuration, but based on the second item in each pair (sum of digits).

The parent(), child_left() and child_right() methods all take an index and return an index for the parent node, left child node, and right child node respectively that will correspond to either the mNodes vector or mSumNodes vector.
Parent node index formula for a heap vector configuration is (index - 2) / 2.
Right child node index formula for a heap vector configuration is (2 * index + 2).
Left child node index formula for a heap vector configuration is (2 * index + 1).

The heapify_down_mNodes() and heapify_down_mSumNodes() both take a node index *i* and use the child_right() and child_left() methods to get the *ith* node's child node indexes. Two comparisons are done to see which of the 3 nodes (*i*, left child, right child) is the maximum. If the current node at index *i* is not the maximum then it will be swapped with the maximum child node, the method is called recursively on the current node until it is greater than both its children. The difference between heapify_down_mNodes() and heapify_down_mSumNodes() is, heapify_down_mNodes() will compare and swap nodes in the mNodes vector, using the first item in each pair (value) for comparison, whereas heapify_down_mSumNodes() will compare and swap nodes in the mSumNodes vector, using the second item in each pair (sum of digits) for comparison.

The heapify_up_mNodes() and heapify_up_mSumNodes() both take a node index *i* and use the parent() method to get the parent node index of *i*. The current node is then compared with it's parent, if the parent is less than the current node then the nodes will be swapped and the method will be called recursively on the current node (note that the current node is now at the

parent nodes index, therefore the method will be called recursively on the parent index). The difference between heapify_up_mNodes() and heapify_up_mSumNodes() is, heapify_up_mNodes() will compare and swap nodes in the mNodes vector, using the first item in each pair (value) for comparison, whereas heapify_up_mSumNodes() will compare and swap nodes in the mSumNodes vector, using the second item in each pair (sum of digits) for comparison.

The top_max_value() method will return the root node's (at index 0) first pair item (value) of the mNodes vector.
Similarly the top_max_sum() method will return the root node's first pair item of the mSumNodes vector.

The pop_max_value() method is used to pop the top node from the mNodes vectors, but we must also pop the same value in the mSumNodes vector to avoid player 2 choosing a value that has already been picked by player 1. The node at the top of mNodes may not be at the top of mSumNodes, therefore a linear search will be used to find the node's index in mSumNodes, once we have the index we can consider that node to be the root, replace it with the last leaf node and call heapify_down_mSumNodes() on that node to correctly order the mSumNodes vector.
We then replace the top value from mNodes (at index 0) with the last leaf node of mNodes and call heapify_down_mNodes() on that node.

The pop_max_sum() method is the same as the pop_max_value() method, the difference is operations on each vector are swapped.

The push() method simply takes a value as a parameter, calculates the sum of digits for that value then forms a pair of (value, sum). The pair is placed at the end of mNodes and heapify_up_mNodes() is called for that node. Similarly the pair is placed on the back of mSumNodes and heapify_up_mSumNodes() is called for that node.

The data is loaded from an input file into a linked list of Test Cases, where a Test Case is an object containing the required information for a single test case, which includes the maximum number of turns, the priority queue of values and who's current turn it is. This process is done in the get_file_data() function.

The max_score_solver() function takes a single test case as a parameter. It runs a loop while the priority queue is not empty. In this loop another loop runs for *k* (max turns) times where, depending on who's turn it is, either the top_max_value() or top_max_sum() value is popped from the queue and added to the player's score.

Pseudocode

```
BEGIN main_program
        // testCases : linked list of Test Cases
        testCases ⇦ get_file_data(“input.txt”)
        For each testCase in testCases:
                max_score_solver(testCase)
 END

BEGIN max_score_solver
        // Input: testCase
        scottScore ⇦ 0
        rustyScore ⇦ 0
        priorityQueue ⇦ testCase.PriorityQueue
        maxTurns ⇦ testCase.maxTurns

        WHILE testCase.priorityQueue size > 0:
                // Check if it is Scott’s or Rusty’s turn
                IF testCase.isTails:
                        // Rusty’s turn
                        // Run for max turns
                        FOR turn ⇦ 0 to maxTurns - 1 and not priorityQueue.empty():
                                // Get the top value for rusty (ball with maximum sum of digits)
                                rustyScore ⇦ rustyScore + priorityQueue.top_max_sum()

                                priorityQueue.pop_max_sum() // pop value from queue

                ELSE:
                        // Scott’s turn
                        // Run for max turns
                        FOR turn ⇦ 0 to maxTurns - 1 and not priorityQueue.empty():
                                // Get the top value for scott (ball with maximum value)
                                scottScore ⇦ scottScore + priorityQueue.top_max_value()

                                priorityQueue.pop_max_value() // pop value from queue

                // Change player
                testCase.isTails ⇦ not testCase.isTails

        // Print the results
        Print scottScore
        Print rustyScore
END
```

Pseudocode for the Priority Queue Class

mNodes ⇦ vector of pairs of integers
mSumNodes ⇦ vector of pairs of integers

```
BEGIN PriorityQueue.push
        // Input: Integer value

        temp  ⇦  value
        // Calculate sum of digits of value
        WHILE temp is not 0:
                sum ⇦ sum + temp mod 10
                temp ⇦ temp / 10

        // Push the node pair to each vector and call heapify up to bubble it up to its position

        mNodes.emplace_back(value, sum)
        heapify_up_mNodes(mNodes size - 1)

        mSumNodes.emplace_back(value, sum)
        heapify_up_mNodes(mSumNodes size - 1)

END
```

```
BEGIN PriorityQueue.top_max_value

        IF mNodes is empty THEN throw error
        ELSE RETURN mNodes[0].first
END
```

```
BEGIN PriorityQueue.top_max_sum

        IF mSumNodes is empty THEN throw error
        ELSE RETURN mSumNodes[0].first
END
```

```
BEGIN PriorityQueue.pop_max_value
        IF mNodes is empty THEN throw error
        ELSE:
                // Get the top node of mNodes
                node ⇦ mNodes[0]

                // Find the node's index in mSumNodes
                idx ⇦ findIndex(node, mSumNodes)

                // Consider idx to be the root node of mSumNodes and replace the root node with
                // the last leaf node and heapify down from there
                mSumNodes[idx] ⇦ mSumNodes.back()
                mSumNodes.pop_back()
                heapify_down_mSumNodes(idx)

                // Replace the last leaf node of mNodes with the root node of mNodes and
                // heapify down from there
                mNodes[0] ⇦ mNodes.back()
                mNodes.pop_back()
                heapify_down_mNodes(0)
END

BEGIN PriorityQueue.pop_max_sum
        IF mSumNodes is empty THEN throw error
        ELSE:
                // Get the top node of mSumNodes
                node ⇦ mSumNodes[0]

                // Find the node's index in mNodes
                idx ⇦ findIndex(node, mNodes)

                // Consider idx to be the root node of mNodes and replace the root node with
                // the last leaf node and heapify down from there
                mNodes[idx] ⇦ mNodes.back()
                mNodes.pop_back()
                heapify_down_mNodes(idx)

                // Replace the last leaf node of mSumNodes with the root node of mSumNodes
                // and heapify down from there
                mSumNodes[0] ⇦ mSumNodes.back()
                mSumNodes.pop_back()
                heapify_down_mSumNodes(0)
END
```

```
BEGIN PriorityQueue.heapify_up_mNodes
        // Input: index of a node ( idx )

        parent_idx ⇦ parent(idx)

        // If the node's value is greater than its parent's then swap values and recurse
        IF mNodes[idx].first > mNodes[parent_idx].first:
                swap(mNodes[idx], mNodes[parent_idx])
                heapify_up_mNodes(parent_idx)
END
```

```
BEGIN PriorityQueue.heapify_up_mSumNodes
        // Input: index of a node ( idx )

        parent_idx ⇦ parent(idx)

        // If the node's sum of digits is greater than its parent's then swap values and recurse
        IF mSumNodes[idx].second > mSumNodes[parent_idx].second:
                swap(mSumNodes[idx], mSumNodes[parent_idx])
                heapify_up_mSumNodes(parent_idx)

        // If the sums of digits are equal and the node's value is greater than its parents, swap
        IF mSumNodes[idx].second == mSumNodes[parent_idx].second:
                IF mSumNodes[idx].first > mSumNodes[parent_idx].first:
                        swap(mSumNodes[idx], mSumNodes[parent_idx])
END
```

```
BEGIN PriorityQueue.heapify_down_mNodes
        // Input: index of a node ( idx )

        // Get child node indexes
        l_idx ⇦ child_left(idx)
        r_idx ⇦ child_right(idx)
        max_idx ⇦ idx

        // Compare the node's value with its children's values
        IF mNodes[ l_idx ].first > mNodes[ idx ].first:
                max_idx ⇦ l_idx
        ELSE IF mNodes[ r_idx ].first > mNodes[ max_idx ].first:
                max_idx ⇦ r_idx

        // If the node does not have the max value then swap it with the max node and recurse
        IF max_idx is not idx:
                swap(mNodes[idx], mNodes[max_idx])
                heapify_down_mNodes(max_idx)
END
```

```
BEGIN PriorityQueue.heapify_down_mSumNodes
        // Input: index of a node ( idx )

        // Get child node indexes
        l_idx ⇦ child_left(idx)
        r_idx ⇦ child_right(idx)
        max_idx ⇦ idx

        // Compare the node's sum of digits with its children's sum of digits
        IF mSumNodes[ l_idx ].second > mSumNodes[ idx ].second:
                max_idx ⇦ l_idx

        // If the sum of digits are equal, compare the node's values
        IF mSumNodes[ l_idx ].second == mSumNodes[ idx ].second:
                IF mSumNodes[ l_idx ].first > mSumNodes[ idx ].first:
                        max_idx ⇦ l_idx

        // Compare the node's sum of digits with its children's sum of digits
        IF mSumNodes[ r_idx ].second > mSumNodes[ max_idx ].second:
                max_idx ⇦ r_idx

        // If the sum of digits are equal, compare the node's values
        IF mSumNodes[ r_idx ].second == mSumNodes[ max_idx ].second:
                IF mSumNodes[ r_idx ].first > mSumNodes[ max_idx ].first:
                        max_idx ⇦ r_idx

        // If the node does not have the max sum of digits then swap it with the max node and
        // recurse
        IF max_idx is not idx:
                swap(mSumNodes[idx], mSumNodes[max_idx])
                heapify_down_mSumNodes(max_idx)
END
```

```
BEGIN PriorityQueue.parent
        // Input: index of node ( idx )
        // Index of parent node is given by the formula below
        RETURN ( idx - 1 ) / 2
END
```

```
BEGIN PriorityQueue.child_right
        // Input: index of node ( idx )
        // Index of right child node is given by the formula below
        RETURN ( idx -* 2 + 2 )
END
```

```
BEGIN PriorityQueue.child_left
        // Input: index of node ( idx )
        // Index of left child node is given by the formula below
        RETURN ( idx * 2 + 1 )
END
```

Results

Input (Test Case Number)	Output (Scott Score Rusty Score)	CPU Time (seconds)
1	1000 197	0.000007
2	240 150	0.000001
3	2100000000 98888899	0.000001
4	9538 2256	0.000002
5	30031 17796	0.000011
6	4726793900 3941702128	0.000003
7	13793 12543	0.000004
8	2195 1643	0.000009
9	3923529875 3049188235	0.000002
10	0 284401	0.000005

It can be inferred from the results table above that the algorithm produced the correct output for the given input data and did so efficiently. A more in depth analysis is done in the section below.

Correctness of Algorithm - Proof by Contradiction

The problem states that both players want to maximise their score. An optimal solution is produced when both players select a ball in an order that optimises their score based on their idea of maximum. This can be achieved using a greedy selection in the form of a priority queue.

The following will prove the correctness of the algorithm for both cases of Rusty and Scott. *Note: we are already aware that the functionalities of the heap operations are correct.*

Scott's Case:
Consider a vector of balls that represents the selection order for Scott of b_1, b_2, b_3,...,b_n. The solution for Scott to optimise his score follows the greedy approach and orders these balls in descending order based on their values in the form of a priority queue.

Assume that Scott's selection does not lead to an optimal solution/score for him. This suggests that the selection vector contains a parent-child relationship (b_{Parent} , b_{Child}) such that $b_{Parent} < b_{Child}$.

This change in order contradicts the requirements of a max-heap priority queue and therefore, can never happen in my priority queue implementation.

Rusty's Case:
For Rusty's case, also consider a vector of balls representing the selection order of b_1, b_2, b_3,...,b_n. The solution for Rusty to optimise his score also follows the greedy approach but instead orders these balls in descending order based on their sum of digits, and if the sum of digits is equal, then based on their value, in the form of a priority queue.

Assuming Rusty's selection does not lead to an optimal solution/score for him suggests that the selection vector contains a parent-child relationship (b_{Parent} , b_{Child}) such that SumOfDigits(b_{Parent}) < SumOfDigits(b_{Child}). The non-optimal selection may also suggest the case of a parent-child relationship (b_{Parent} , b_{Child}) such that SumOfDigits(b_{Parent}) = SumOfDigits(b_{Child}) AND $b_{Parent} < b_{Child}$.

My priority queue implementation for Rusty uses the sum of digits as the comparator values, and if the sum of digits are equal, it will use the ball values as the comparator values.
Therefore the non-optimal case contradicts my priority queue implementation for Rusty and therefore would never occur in my algorithm.

Performance Analysis

The performance analysis will be done on each function/method starting with the Priority Queue and finishing with the max_score_solver function which will be combined to give the final Time and Space complexity of the entire program.
Note that both heapify down methods do the same operations and therefore only one of the method's complexities need to be calculated. The same occurs for the heapify up methods and the pop method.

Note that heapify_down do the same operations and therefore only one of the method's complexities need to be calculated. The same occurs for heapify_up_mNodes and and heapify_up_mSumNodes.

Let n = number of values (balls).
Let k = maximum number of turns.
Let d = maximum number of digits of all the values.

PriorityQueue Space Complexity:

The priority queue object stores two vectors of length *n*, in big O notation 2n is represented as O(n) space. The space complexity of each method is calculated below.

PriorityQueue heapify_down()

Time Complexity:

In the worst case heapify down is called for the root node, each recursive call will either be on the left or right child of the current node, meaning halving the vector on each recurse. All other operations take constant time. We can represent the time complexity of this method from the last recursive call to the first with the recurrence relation T(n) = T(n/2) + 1, where T(1) = 1 is the base case.

$T(n) = T(n/2) + 1$
$= T(n/4) + 2$
$= T(n/8) + 3$
$= T(n/(2^v)) + v$
$= 1 + \log_2 n$

$T(n/(2^v)) = T(1)$
$n/(2^v) = 1$
$2^v = n$
$v = \log_2 n$

Therefore the time complexity for heapify down by taking the dominant term is O(logn)

Space Complexity:

The heapify down method takes constant space O(1) expect for the recursive call, as calculated above the recursive call runs for $\log_2 n$ times, as each function call is store on the stack we get the space of O(1) * $\log_2 n$ which gives us O(logn) space complexity.

PriorityQueue pop()

Time Complexity:

The first operation of the pop() method is to find the location of a node in the opposing vector (either mNodes or mSumNodes). This operation is a linear search and in the worst case the node will be at the end of the vector, therefore this operation runs O(n) times. However, if a node is found at the end of the vector and we get O(n) time then the node is the last leaf node, meaning heapify_down() will not recur and have a constant time of O(1). We will use the worst case in this scenario if the node we are looking for is the last leaf node as O(n) is greater than O(logn).

The second heapify down called in pop() is always done at the root node and therefore will always have a time of O(logn). The final complexity of this method is O(logn) + O(n). The dominant term is O(n), therefore making the time complexity of pop() O(n).

PriorityQueue heapify_up()

Time Complexity:

The process of the heapify up is similar to heapify down but in reverse, so we start at the child and traverse to the root node at worst case. Therefore we can also write the recurrence relation as $T(n) = T(n/2) + 1$, where $T(1) = 1$ is the base case. From the calculation in the heapify down section we get the time complexity of O(logn).

Space Complexity:

The space complexity is also the same as the heapify down space complexity.
O(logn).

PriorityQueue push()

Time Complexity:

The first part of this method calculates the sum of digits of the given value, at the worst case this takes *d* (max number of digits) time to compute O(d). We then call heapify twice for each vector (mNodes and mSumNodes). This results in the total complexity of O(d) + O(logn) + O(logn). *d* is not proportional to *n* which means our final time complexity for push() is O(logn + d).

max_score_solver()

Time Complexity:

The max_score_solver method has one main loop that will run until the priority queue is empty i.e. *n* times. All while loops nested in this loop will break when the loop is empty, therefore these loops do not need to be included in the complexity. In the while loop there are two pop() methods, therefore the max solver time complexity will equal O(n) * O(n) which is $O(n^2)$ time complexity worst case. However this worst case would only occur if the a value in an opposing vector in the priority queue was always the last leaf node, which is very unlikely. If we say on average that a value to be popped is at the middle index of the opposing vector we would get a complexity of $n/2 + (\log_2 n)/2$ as we would be running the linear search for n/2 times and heapify_down $(\log_2 n)/2$ times.

Therefore we get
$O(n^2)$ worst case with a more likely time complexity of $n/2 + (\log_2 n)/2$ time which equals
$O(n + \log_2 n)$.

get_file_data()
Time Complexity:
The get_file_data method's main operation is to push values onto the priority queue, this push operation occurs *n* times, all other operations occur in constant time. The push operation has a time complexity of $O(\log_2 n + d)$. Therefore, the final time complexity of get_file_data for a single test case is $O(\log_2 n + d) * O(n)$ which gives us $O(d\log_2 n + nd)$ time complexity.

Performance conclusion:
Now that we have calculated the complexity for each method we take the most dominant part that is proportional to the input.

Time Complexities:
For $O(\log_2 nd + nd) + O(n^2)$,
$O(n^2)$ is the dominant term, therefore,
Final Worst Case Time Complexity: $O(n^2)$

For $O(\log_2 nd + nd) + O(n + \log_2 n)$,
$O(\log_2 nd + nd)$ is the dominant term, therefore,
Final Average Time Complexity: $O(\log_2 nd + nd)$

Space Complexity:
Final Space Complexity: $O(\log_2 n)$.