2801ICT Assignment 2 – Maximise The Score

S5084207

Yoon Jin Park

Bachelor of Computer Science

Griffith University

Introduction

The purpose of this report is to outline the design and pseudo codes of the algorithm to solve "Maximise The Score" games and provide results of the algorithm while analysing the algorithm's correctness and performance.

"Maximise The Score" game is to make the best total score for each player based on their priority. Scott selects the maximum value based on the value written on a ball while Rusty choose the maximum value based on the sum of digits of the values on a ball. Considering their priorities, the objective of the game is to maximise the sum of the chosen balls for each player. If the result of a coin toss is "HEADS", Scott starts first for the maximum turn allowed while Rusty starts first for the maximum turn allowed if the result of a coin toss is "TAILS".

This report is structured as follow. Algorithmic design encompassing the overview of the algorithm, details of the description and associated pseudo codes are presented in the next section. This then is followed by the result and algorithm analysis, including its correctness and performance analysis.

Algorithmic Design

Overview of the algorithm

The main components of the "Maximise the score" algorithm can be described with the following three aspects: Game Ball, Priority queue implemented with a binary heap and Score Maximiser. These main components, implemented as classes, have specific roles.

First, the Game Ball class is responsible for holding the value written on a ball, calculating the sum of digits and storing this value, and setting and storing the state of the ball as to whether it is taken or not.

Second, the Priority Queue class mainly performs functions of a priority queue based on a heap and providing the max value based on a different priority criterion set for each player. This class has usual methods for a priority queue, such as front, insert, delete, and is_empty. The details of this class's roles are described in Algorithm Description section.

Lastly, the Score Maximiser class is responsible for building two different priority queues and dealing with playing rounds for players. In addition, this class calculates the final score and returns the sum to a main function.

Algorithm Description

This section addresses what methods are implemented in each of the three classes introduced in the overview section and provides the rationale behind these implementations.

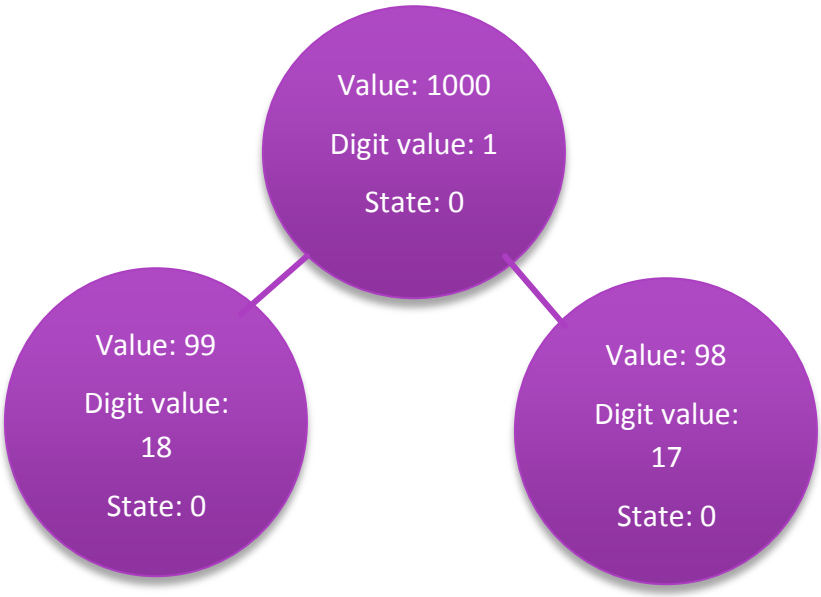
The algorithm takes the first input, the number of games/test-cases. This represents how many games will be played for this program run. After this first line, every three-line provides the information for each game. This consists of: 1) the number of balls on a table; 2) the maximum number of turns per round; 3) balls with the value; and 4) the result of the toss determining which player starts first. These data are passed to a dictionary, which then are stored in different variables to be fed to the Game Ball, Priority Queue, and Score Maximiser class.

The output of the algorithm provides the maximised score for each player. This is printed out on the terminal in addition to generating the output file in a text format.

The main driver function initiates each game in the number of games. In the main driver function, the balls for each game are stored with the value written on a ball, the sum value of digits, and the state of taken (initialised as not taken i.e. 0). Calculating the sum of digit values is implemented as part of the Game Ball class since it needs to be calculated only once. The main function then instantiates the Score Maximiser class that builds two heaps for each player, plays rounds, and produces the output file that documents the result of each game.

The instantiated Score Maximiser class then builds two priority queues: one for Scott, the other for Rusty. Each priority queue inserts the balls from the same list, however, with a different priority criterion. For Scott, the priority is the value itself while the priority is the sum of digit values for Rusty. The priority queue is implemented through a max heap, thus, the maximum value will exist at the front of the queue. Since the queue is based on the heap, the queue is not sorted perfectly. However, through a heapify function, each time the maximum ball is extracted from the front. The example of a priority queue initialisation is illustrated below.

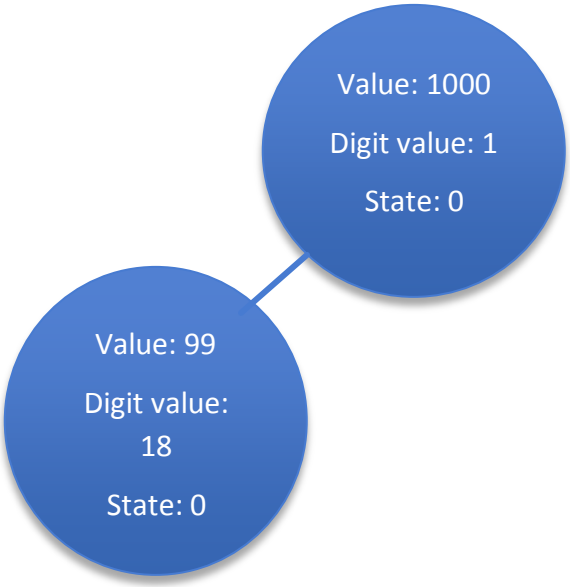
Figure 1. Example of the initialised priority queue with the balls [1000, 99, 98] for Scott



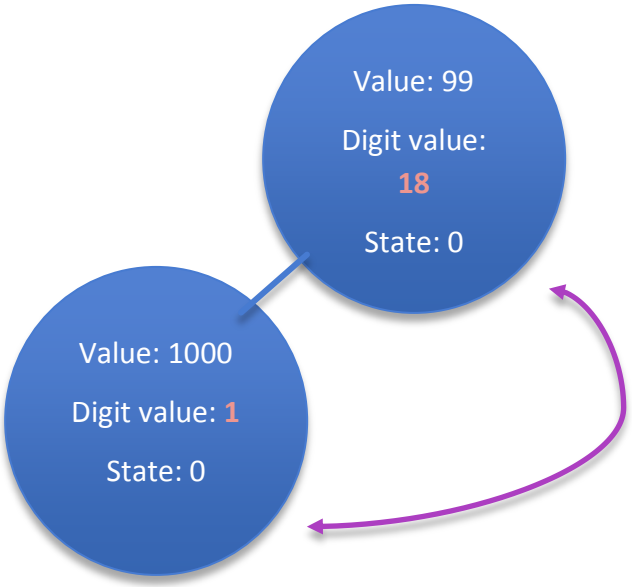
The priority queue does not only append the balls at the end. Once it is appended, the queue reorders it through a heapify method. In this particular case, the heapify up is implemented to ensure the queue does not violate the heap property where a parent node is expected to have a larger value than child nodes. Especially, based on the priority, for example, for Rusty, if a ball inserted has a larger value than its parent nodes, a ball moves up until the heap property is satisfied. The order on the same level in the heap does not matter. However, if multiple balls have same priority, that is the same sum of digits, then the ball with a higher value must be having a higher priority than the other balls. The example of the heap construction at the initialisation for Rusty is described in Figure 2.

Figure 2. Example of the initialised priority queue construction process with the balls [1000, 99, 98] for Rusty

a. Initialise a root of the first ball and insert the second ball

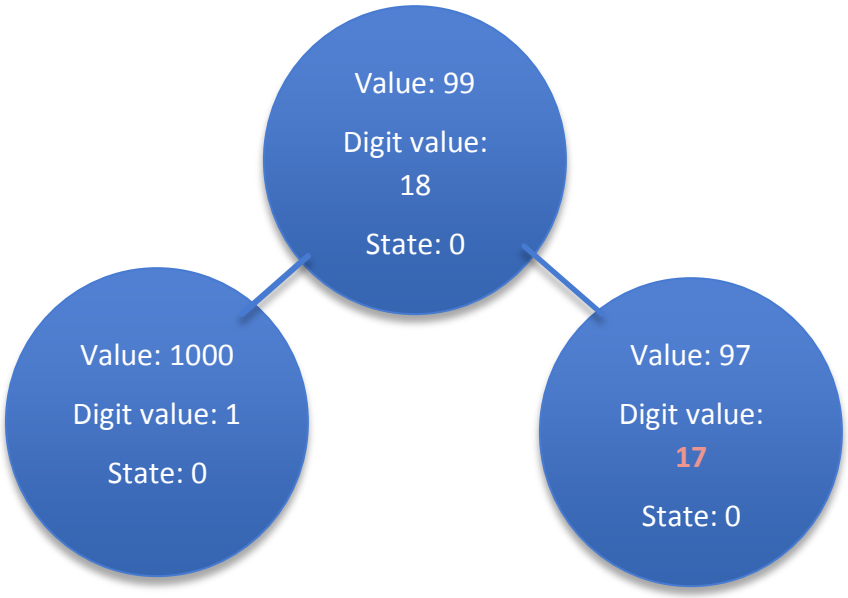


b. Heapify up because the priority of the second ball has a higher priority (i.e. Digit value 18 > Digit value 1)



Heapify up (swap two values based on their priorities)

c. Insert the third ball and check the priority – no violation of the heap properties

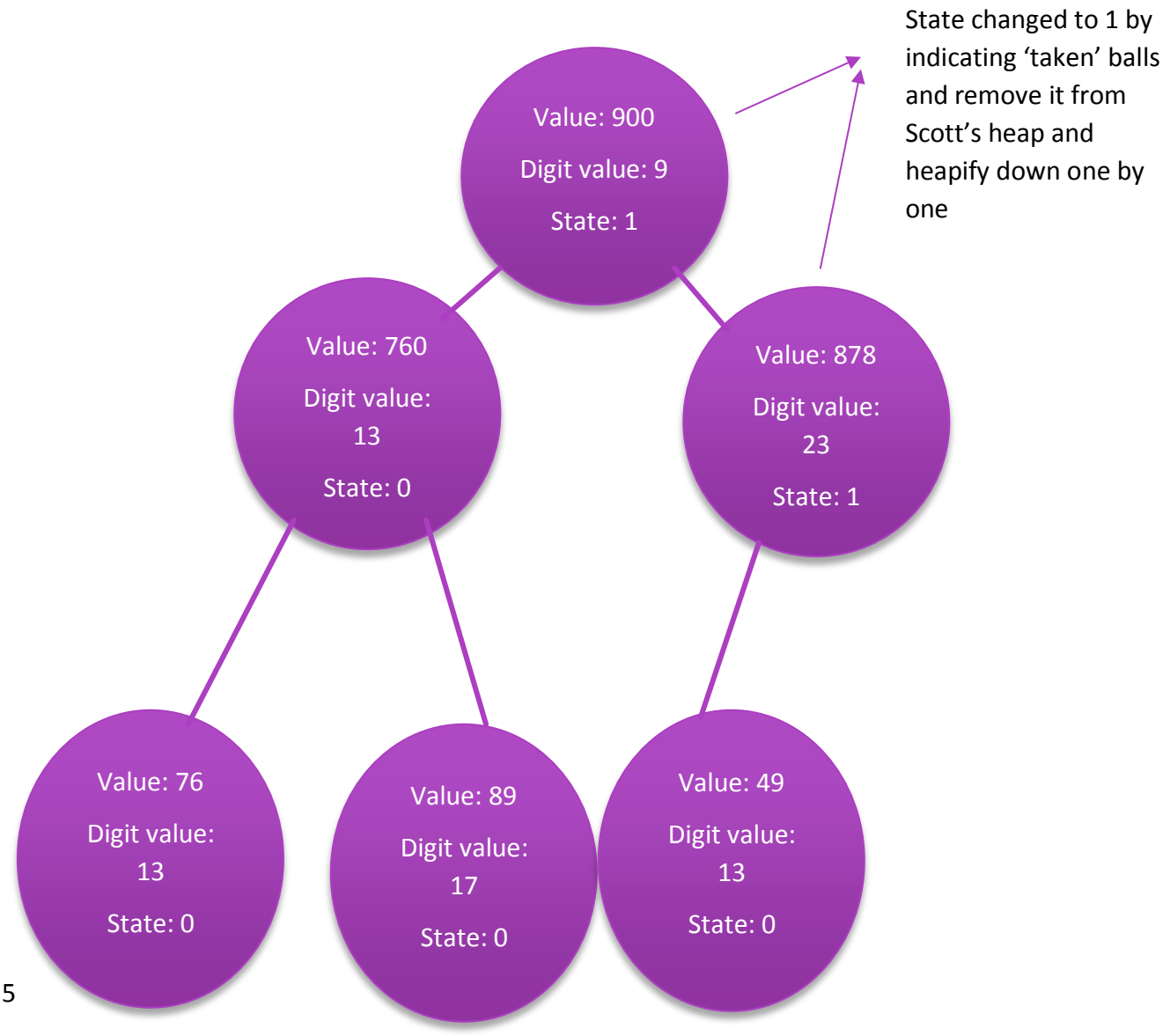


After the game is started, the core algorithm methods are used: play round method from the Score Maximiser class and delete method from the Priority Queue class. First, play round method calls each player's round in the game. The number of rounds is determined by taking ceiling number of the number of balls divided by the maximum turns allowed. For each round in the number of rounds, depending on whose turn it is, the balls are taken based on the priority. During this process, the number of balls taken is tracked so that when there is no ball left, the algorithm terminates.

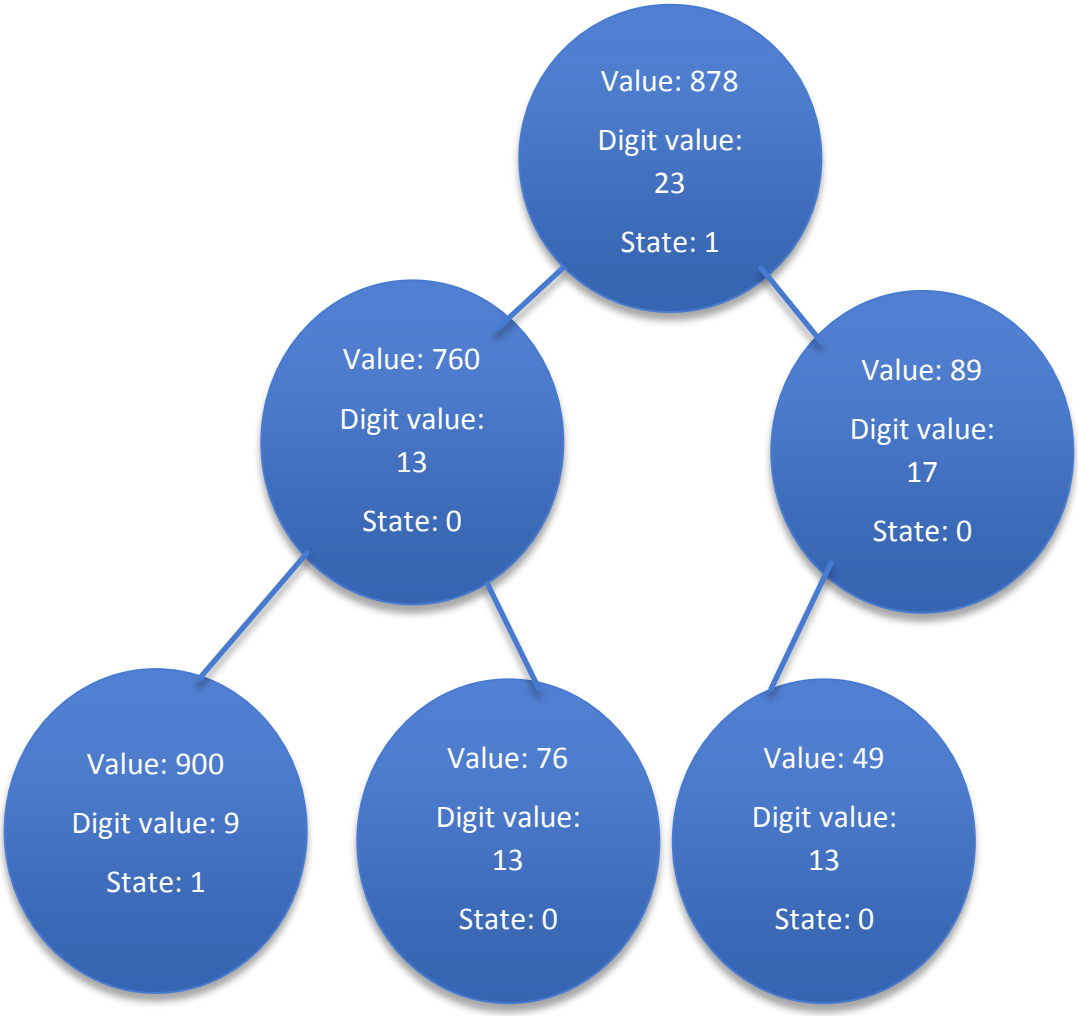
Second, delete method is called from play round method described above. This method handles critical steps in the algorithm. The method takes the priority (either value or the sum of digits) and the maximum number of turns allowed. Then, it checks whether the ball has been taken by the other player by examining the state of the ball and if it is not taken, extract the ball and delete it by setting the state to 1 (i.e. taken) and heapify the heap. This happens for the maximum number of turns allowed. If the front ball is taken by having the state 1, then this method deletes the balls and heapify it until it reaches the ball that does not have the state 1. During this process, the turns are not counted because the player did not extract any ball yet. Once it finds the maximum ball that is not deleted, it starts to count the turns. The following example describes this situation of a change of the ball state, deletion, heapify down, and not counting the turn when the ball at the front queue is taken.

Figure 3. Select the maximum for Scott and Rusty with the ball list [76, 89, 49, 900, 760, 878], with the maximum turns of 2 and Heads (i.e. Scott selects first)

a. Building a queue for Scott and taking two maximum values by changing the state to 1 and removing them from Scott's heap



b. Rusty's priority queue where two balls has the state of 'taken' and in particular, the front ball is taken -> in this case, remove the first ball and heapify down without counting his turn because Rusty did not take any ball yet



When the maximum turns allowed for each player are done, the turn indicating the player through "HEADS" (i.e. Scott) or "TAILS" (i.e. Rusty) will be toggled and the round for the next player will be started.

Once each player's round is complete, the sum of the values including the values of the balls taken from this round is calculated and stored in the results dictionary.

If the whole balls are taken, the game is finished and the main driver function moves to the next game available until there is no game left.

In the next section, these descriptions of the algorithm are implemented in Pseudo Code.

Algorithm Pseudo-Code

The first core class is Score Maximiser that builds the queues and initiates a game play and returns the final result. Two main methods, build_pq and play_round, are addressed below.

```
class ScoreMaximiser:
//build a priority queue for each player based on one's priority
   Algorithm build_pq(ball_list):
      for i <- 0 to ball_list.length do
         insert balls to scott_pq based on value priority
         insert balls to rusty_pq based on sum of digit priority
//start to play rounds
   Algorithm play_round(maximum_turns, player_turn):
      rounds <- take ceiling of number of balls/maximum_turns
      toggled <- initialise the turn of this round for a player (e.g. "HEADS" then Scott plays first)
      track the number of balls taken from 0
      if the whole number of balls are not taken:
         for game_round <- 0 to rounds do
            if toggled="HEADS":
               max_taken <- call delete(toggled, maximum_turns) on scott_pq
               if max_taken has some value:
                  add max_taken to "scott"'s result
            if toggled="TAILS":
               max_taken = <- call delete(toggled, maximum_turns) on rusty_pq
               if max_taken has some value:
                  add max_taken to "rusty"'s result
            toggled <- toggle the turn so that the next player can play a round
            add maximum_turns to the tracked number of balls to count the balls taken
      return the final score for each player
```

The next essential class is Priority Queue. This class has normal methods, such as is_empty, front, insert, and delete. Since this class is implemented based on a heap, heapify up and heapify down methods are also implemented. The utility methods such as swap and print queue are included in the following Pseudo Code.

```
class PriorityQueue:
    Algorithm is_empty():
        return True if ball_heap .length = 1 //first element in the queue is reserved not to be used
    //return length of the current  queue
    Algorithm length():
        return ball_heap.length
    //return the front value i.e. value of the max ball which is in the queue's second index
    Algorithm front():
        return ball_heap[1].get_value()
    //return the ball object in the queue's second index i.e. max ball
    Algorithm front_index():
        return ball_heap[1]
    //util function swap – swapping two balls x and y
    Algorithm swap(x, y):
        temp = ball_heap[y]
        ball_heap[y] = ball_heap[x]
        ball_heap[x] = temp
    //util function – print the value, the sum of digits, and the state of each ball in the queue
    Algorithm print_queue():
        for i <- 0 to ball_heap.length do
            print out "value: ", ball_heap[i].get_value(),
                " digit sum: ", ball_ queue [i].get_digit(), " state: ", ball_ queue [i].get_state())
```

The insert method is important to build a priority queue based on each player's priority. First, append the ball from the list and increase the size of the queue and call heapify up to have the maximum priority to be at the front of the queue.

```
//insert the ball and build the heap, followed by heapify
Algorithm insert(ball_picked, priority):
   ball_heap.append(ball_picked)
   size += 1
   heapify_up(size, priority)
```

The heapify up and heapify down methods are implemented in consideration of different priorities for Scott (HEADS) and Rusty (TAILS). The method considers the max heap property.

```
//heapify  up makes the ball goes up to meet the max heap property where a parent node must
be larger than child nodes.
Algorithm heapify_up(size, priority)
      while floor value of size of the queue/2 > 0 do
        if priority="HEADS" //Scott's priority
          //parent node < child node then swap based on the value
          if ball_heap[size//2].get_value()<ball_heap[size].get_value()
             swap(size//2, size)
        if priority="TAILS" //Rusty's priority
          //parent node < child node then swap based on the sum of digits
          if ball_heap[size//2].get_digit()<ball_heap[size].get_digit()
             swap(size//2, size)
          //parent node = child node then look at the value and swap based on the value
          if ball_heap[size//2].get_digit()=ball_heap[size].get_digit()
             if ball_heap[size//2].get_value()<ball_heap[size].get_value()
                swap(size//2, size)
        //reduce index size to half to do the same process until the max heap property is met
        size = size//2
```

```
//heapify down method makes a parent node that is smaller than child nodes to be placed at the right position in the queue
  Algorithm heapify_down(c_index, priority)
    //while current index is smaller than the size of the queue
     while (c_index * 2) <= size of the queue do
       //get the child node with a larger value based on the priority
       max_child_index = get_max_child(c_index, priority)
       if priority="HEADS" //Scott's turn
         //if the value of the current index < the value of the max_child_index then swap
         if ball_heap[c_index].get_value() < ball_heap[max_child_index].get_value()
           swap(c_index, max_child_index)
       if priority="TAILS" //Rusty's turn
         //if the sum of digits of the current index < the sum_of digits of the max_child_index then //swap
         if ball_heap[c_index].get_digit() < ball_heap[max_child_index].get_digit()
           swap(c_index, max_child_index)
         //if the sum of digits of the current index = the value of the max_child_index
         if ball_heap[c_index].get_digit() == ball_heap[max_child_index].get_digit()
           //consider the value of the current index compared to the value of the max_child_index
           if ball_heap[c_index].get_value() < ball_heap[max_child_index].get_value()
             swap(c_index, max_child_index)
       //current index goes down to the max_child_index to meet max heap property
       c_index = max_child_index
```

The most important method to play turns is the “delete” method that deals with extracting the max ball based on the player’s priority and heapify the queue. This method also enables the player to ignore the balls that are already taken.

```
Algorithm delete(priority, turns)
  //initialise the sum of the max balls for each player in this round
   max_addition = 0
  //track the number of turns taken by each player
   player_turn = 0
  //if the player did not exhaust his turn, keep extracting the max ball
   while player_turn != turns
      //find the max and change deleted state to 1
      if front_index().get_state() is not taken
         max_deleted = front() //get the max ball
         ball_heap[1].set_state_deleted() //change the state
         //remove the element by assigning the least value and reduce the size of the heap
         ball_heap[1] = ball_heap[size]
         size -= 1
         //heapify down to ensure the heap property is still met
         heapify_down(1, priority)
         //add the extracted ball to the result of this round
         max_addition += max_deleted
         //the player used one turn
         player_turn += 1
      else
         //remove the element by swapping and reduce the size of the heap
         ball_heap[1] = ball_heap[size]
         size -= 1
         //heapify down to ensure the heap property is met
         heapify_down(1, priority)
         //but do not add any value to the result or player’s turn as nothing happened
   return max_addition
```

The base class Game Ball is important because it stores the value written on a ball, the sum of digits, and the state whether the ball is taken. The class itself has only a small number of the methods, but they are essential for the priority queue.

```
class GameBall:
    //getters: get the value of the ball, the sum of digits of the ball, and the state of taken
    Algorithm get_value()
        return value
    Algorithm get_digit()
        return digit_value
    Algorithm get_state()
        return state
    //set the state of the ball to deleted (1) when it is extracted
    Algorithm set_state_deleted()
        If state is not 0 //not taken
            state = 1
    //calculate the sum of all digits
    Algorithm cal_digit_sum(input_val)
        if input_val=0
            return 0
        if input_val is not 0
           //call recursive function to add the digit by digit
            return input_val%10 + cal_digit_sum(input_val//10)
```

Result and Algorithm Analysis

In this section, the result of the algorithm and the analysis of this algorithm is presented.

Result

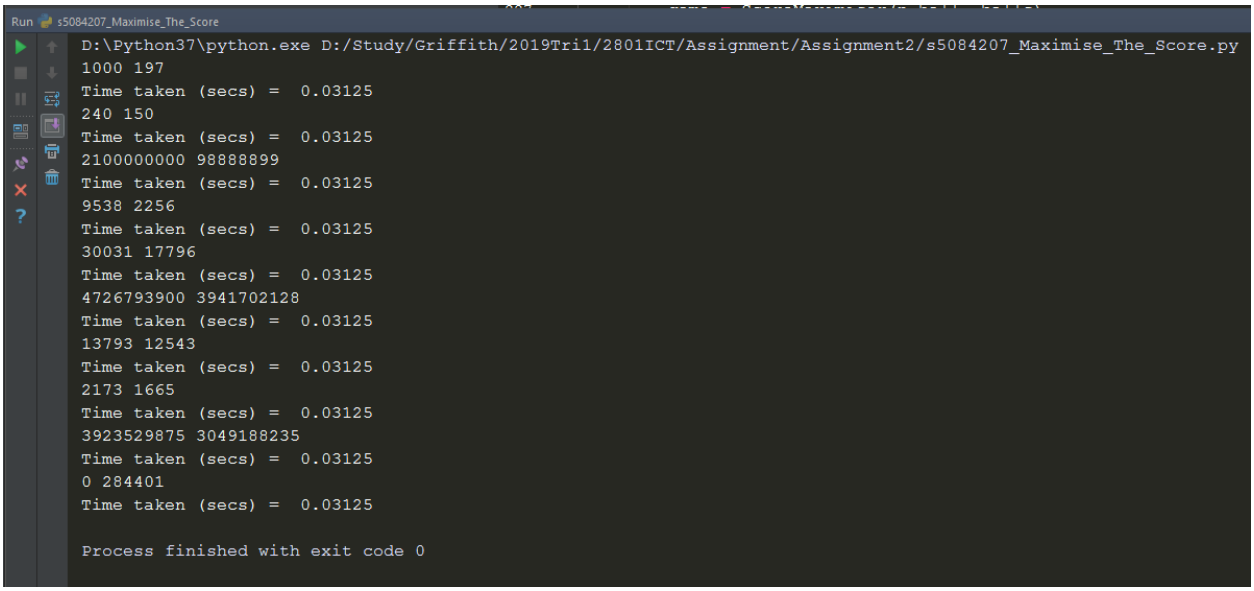
The result to be returned from the algorithm must have two components: total score for each player and CPU time taken.

The output of the algorithm for total score is consistent with the desired output of the input per the assignment instruction document. Table 1 displays the result of the algorithm and CPU time taken in seconds. In particular, CPU time taken is presented as the time taken output subtracting the time taken for running a basic program (i.e. printing out "Hello World").

Table 1. Output and CPU Time for the sample input provided

Input	Output	CPU Time (Secs)	Input	Output	CPU Time (Secs)
1	1000 197	0.000000	**6**	4726793900 3941702128	0.000000
2	240 150	0.000000	**7**	13793 12543	0.000000
3	2100000000 98888899	0.000000	**8**	2173 1665	0.000000
4	9538 2256	0.000000	**9**	3923529875 3049188235	0.000000
5	30031 17796	0.000000	**10**	0 284401	0.000000

Figure 4. Output and time taken screen image – in this case, printing out "Hello world" took 0.03125



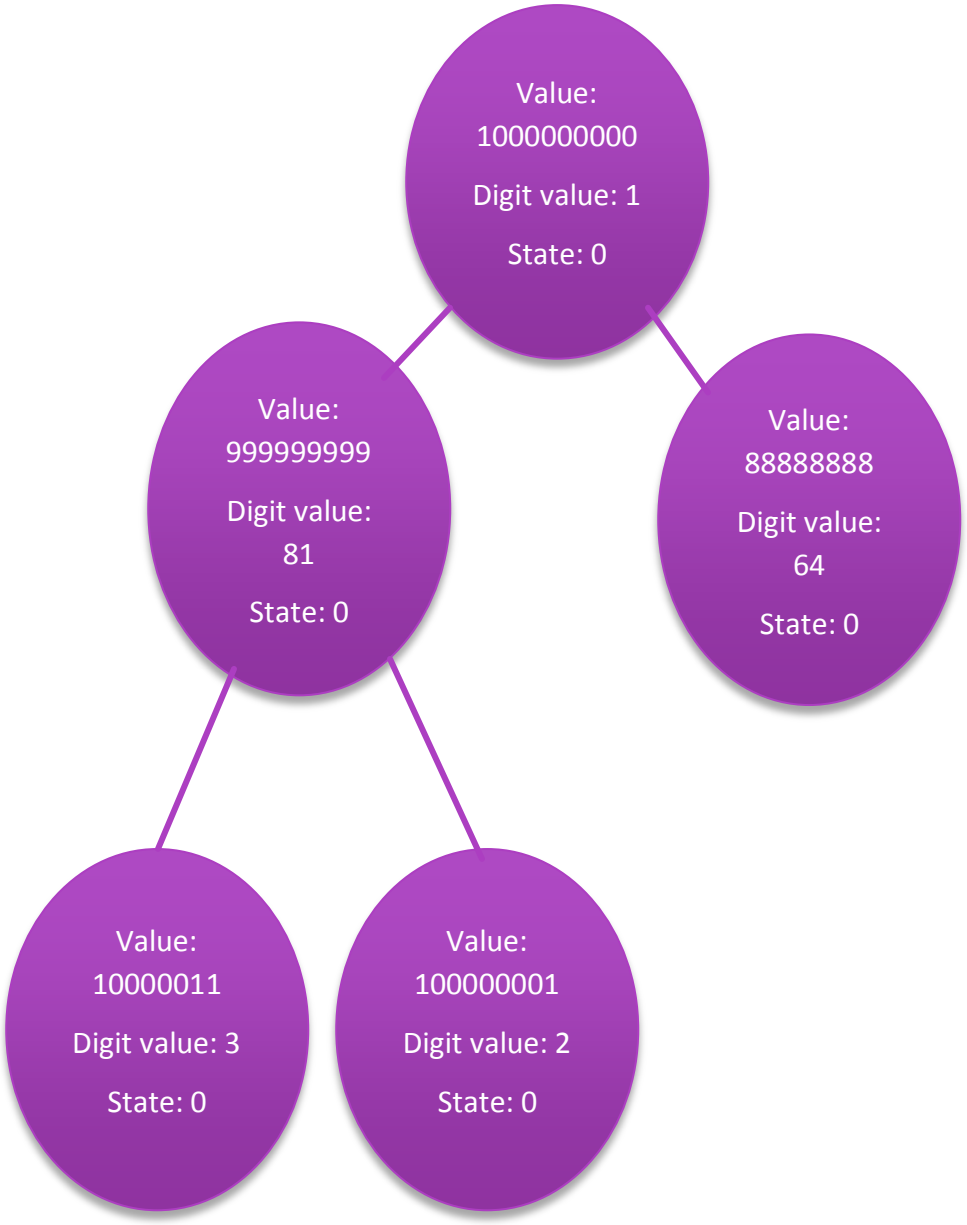
To analyse how the game has been progressed, the third case example from the sample input file is used. There are five balls and the maximum two turns are allowed and Scott starts first in this

game. The example ball list of [88888888 10000011 999999999 100000001 1000000000] is used below.

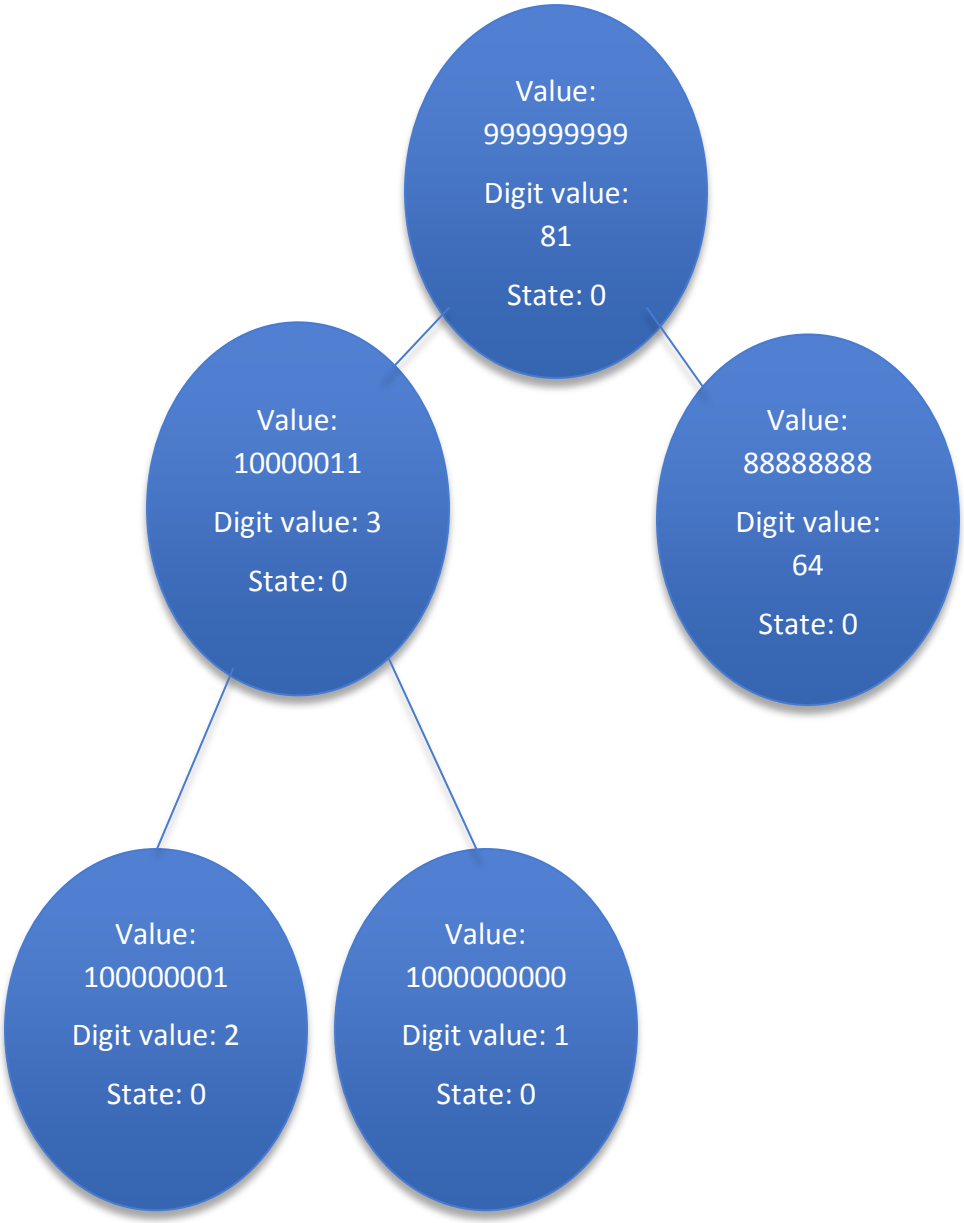
First, with the ball list, two different priority queues are built for Scott and Rusty as below. For Scott, the largest value of 100,000,000 is at the front of the queue while for Rusty, the ball with the largest sum of digit, 999,999,999 is at the front.

Figure 5. Initialised queues for the example ball list

a. Initialised Scott's queue

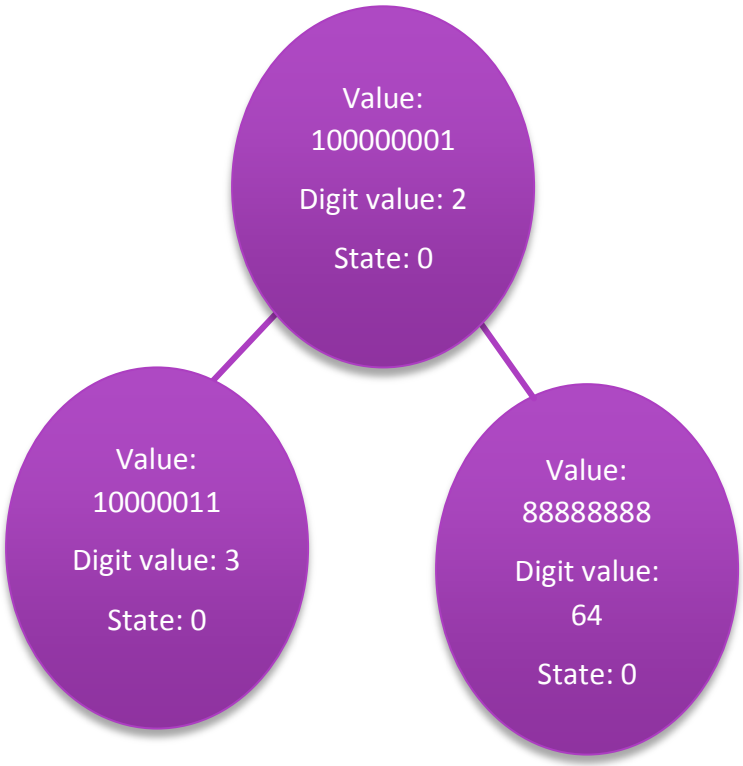


b. Initialised Rusty's queue

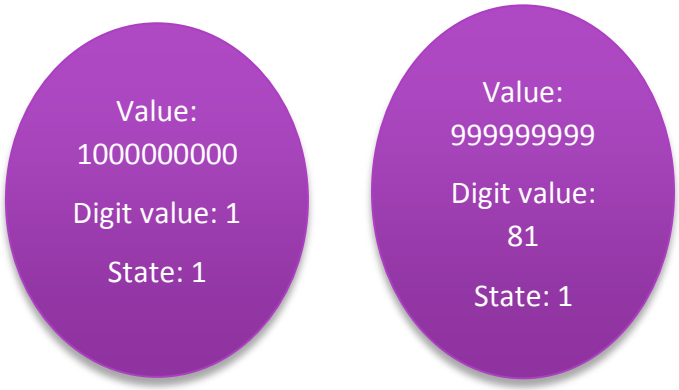


As Scott starts to play first and he has a maximum two turns, two maximum values are taken from the queue. Heapify down is performed at the same time. In this case, adding two max balls taken for Scott results in 1,999,999,999. These two balls now have the state as 1. This is illustrated in Figure 6 below.

Figure 6. a. Result from Scott's round

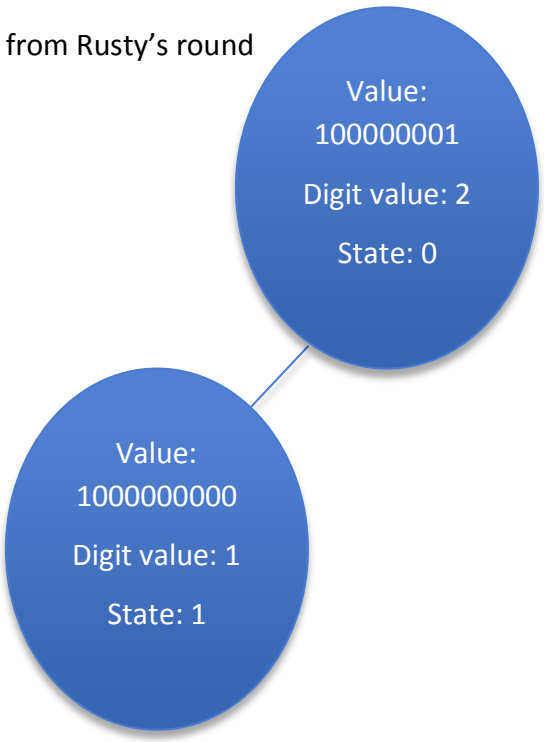


b. Two balls taken for Scott: max_addition = 1,999,999,999



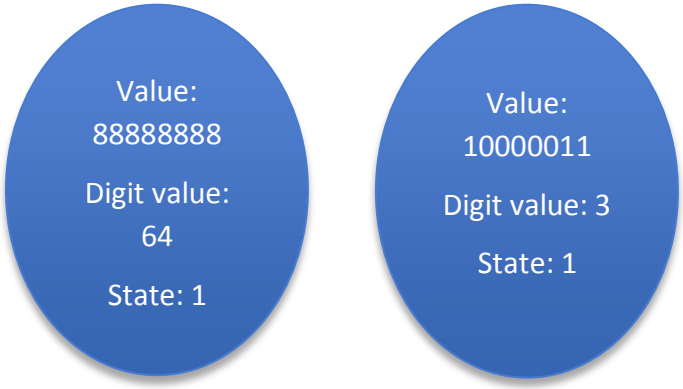
When it becomes Rusty's turn, the ball with the max sum of digits is taken. This ball with the sum of digits, 81, is removed from Rusty's queue without being taken, because it has been picked up by Scott previously. Then, the balls with two next largest sum of digits (88,888,888 and 10,000,011) are selected and the score for this round for Rusty is 98,888,899.

Figure 7. a. Result from Rusty's round



b. Removed from the Rusty's queue because it is taken

c. Two balls taken by Rusty: max_addition = 98,888,899



In Scott's turn, there is only one ball left, thus, Scott takes the last ball that is not taken, adding 100,000,001 to his previous score. This results in 2,100,000,000 total score for Scott. Rusty's total score is still same, 98,888,899.

Algorithm Analysis

Two analysis is discussed in this section: Correctness and Performance. The correctness of the algorithm is attempted to be proved by contradiction. Performance analysis encompasses time and space complexity analysis.

Correctness of the algorithm

Let us assume that the priority queue operation based on a max heap provides the correct output. Assume that there is the optimal selection m* (max ball selection) and the selection from the algorithm m'. Also, assume that there is the input list of 'List of Balls'.

Proof by contradiction: When the ball selection from the algorithm is made, if we assume the current algorithm provides incorrect selection, then maximising score will not happen because Max_Ball is not selected by the algorithm (i.e. m' ≠ m*). This can happen because either:

1) Max_Ball is not an element of List of Balls; or

2) List of Balls has an element, List of Balls[some_index], such that Max_Ball < Ball[some_index].

To prove correctness of the algorithm by contradiction, it is required to prove that these cases are not possible/available, indicating that providing incorrect selection by this algorithm cannot happen.

First, given the assumption that the priority queue operation is perfectly implemented, Max_Ball is initialised to and assigned to elements of List of Balls and inserted into both queues (highlighted in yellow below), so the first case is not possible. Max_Ball must be in List of Balls.

```
for each_ball in range(games[each_game]['n_balls']):
    ball = GameBall(games[each_game]['ball_list'][each_ball])
    balls.append(ball)


Algorithm build_pq(ball_list):
    for i <- 0 to ball_list.length do
        insert balls to scott_pq based on value priority
        insert balls to rusty_pq based on sum of digit priority
```

Secondly, given the assumption that the heapify up and down and other heap operations are implemented perfectly, after some_index iterations of the loop, Max_Ball >= A[some_index] otherwise it would have been swapped in the code lines highlighted in yellow below. Therefore, Max_Ball must be the largest ball going through the perfect heap operation and upon termination of the codes below, Max_Ball is at least as large as every other element in List of Balls or the largest. This is in contrast with the second case listed above where List of Balls has an element, List of Balls[some_index], such that Max_Ball < Ball[some_index]. This case is not possible.

```
Algorithm heapify_down(c_index, priority)
    while (c_index * 2) <= size of the queue do
        max_child_index = get_max_child(c_index, priority)
        if priority="HEADS" //Scott's turn
            if ball_heap[c_index].get_value() < ball_heap[max_child_index].get_value()
                swap(c_index, max_child_index)
        if priority="TAILS" //Rusty's turn
            if ball_heap[c_index].get_digit() < ball_heap[max_child_index].get_digit()
                swap(c_index, max_child_index)
            if ball_heap[c_index].get_digit() == ball_heap[max_child_index].get_digit()
                if ball_heap[c_index].get_value() < ball_heap[max_child_index].get_value()
        c_index = max_child_index
```

Therefore, both first and second case are not possible. It is proved that the algorithm cannot provide incorrect selection from these both cases. Hence, the optimal selection m* and the selection from the algorithm m' must equal (m'=m*) where the algorithm must give the max value based on the priority and maximising score occurs because m'=m*.

Performance analysis

Sine the algorithm enables to build two priority queues from one list of balls, sharing the list of balls makes time and space efficiency better by allowing less work (no need to remove balls each time it is taken by a player) and using smaller space (no duplicate lists of balls required).

Time and space efficiency of the algorithm depends on the two main components – 1) build queues using the insert method that contains a heapify up method, and 2) play rounds using the delete method that includes a heapify down method.

```
Algorithm build_pq(ball_list):
    for i <- 0 to ball_list.length do
      insert balls to scott_pq based on value priority → n
      insert balls to rusty_pq based on sum of digit priority → n
→ O(2n)
Algorithm heapify_up(size, priority)
    while floor value of size of the queue/2 > 0 do
      if priority="HEADS"
        if ball_heap[size//2].get_value()<ball_heap[size].get_value()
          swap(size//2, size)
      if priority="TAILS"
        if ball_heap[size//2].get_digit()<ball_heap[size].get_digit()
          swap(size//2, size)
        if ball_heap[size//2].get_digit()=ball_heap[size].get_digit()
          if ball_heap[size//2].get_value()<ball_heap[size].get_value()
            swap(size//2, size)
      size = size//2
```

$O(2n\log_2 n)$

$\rightarrow O(\log_2 n)$

$O(1)$

First, from the code above, when there are n balls, building two priority queues takes $2n$. During the process of building a queue, an insertion of the ball includes appending the ball takes a constant time $1\ (or\ c)$ and a heapify up method is called.

If we assume a constant time $1\ (or\ c)$ for 1) a comparison to identify the priority ("HEADS" or "TAILS") and 2) a comparison to identify whether the currently appended node has a larger value than its parent's node, 3) swapping operation, and 4) all other division operations, each heapify up method in the worst-case takes $\log_2 n$.

Therefore, the worst-case scenario for building two queues takes $\underline{2 * n * \log_2 n}$ where there are two queues (2) with n balls (n) that need to be a heapify up $(\log_2 n)$ each time.

Next, when the play round method is called with n balls and t maximum turns, if we assume a constant time $1\ (or\ c)$ for 1) a comparison to identify the priority ("HEADS" or "TAILS"), 2) calling the delete method, and 3) all other addition and division operations, the whole play round method takes $\lceil n/t \rceil$. This is then times by the operations occurring in the delete method.

```
Algorithm play_round(maximum_turns, player_turn):
    rounds <- take ceiling of number of balls/maximum_turns
    toggled <- initialise the turn of this round for a player (e.g. “HEADS” then Scott plays first)
    track the number of balls taken from 0
    if the whole number of balls are not taken:
      for game_round <- 0 to rounds do
        if toggled="HEADS":
          max_taken <- call delete(toggled, maximum_turns) on scott_pq
          if max_taken has some value:
            add max_taken to “scott”’s result
        if toggled="TAILS":
          max_taken = <- call delete(toggled, maximum_turns) on rusty_pq
          if max_taken has some value:
            add max_taken to “rusty”’s result
        toggled <- toggle the turn so that the next player can play a round
        add maximum_turns to the tracked number of balls to count the balls taken
    return the final score for each player
```

$\rightarrow O(n/t)$

$\rightarrow O(?)$

$\rightarrow O(?)$

$all\ minor\ operations$
$\rightarrow O(1)$

Thus, the next sub-analysis to be undertaken is on the delete method.

When there is no balls that are taken yet, if we assume a constant time 1 $(or\ c)$ for 1) a comparison to identify whether the max ball is taken and if not taken, change the state to 'taken', 2) swapping the maximum ball with the last ball from the queue and reducing the size of the queue, and 3) all other addition and subtraction operations, then the delete method takes $O(t)$.

However, if there are some balls taken by the previous player, then the worst-case scenario takes $O(2t)$ because the balls taken at the front need to be removed first before the current player can select the balls.

```
Algorithm delete(priority, turns)
    max_addition = 0
    player_turn = 0
    while player_turn != turns                    → O(t) or O(2t)
        if front_index().get_state() is not taken
            max_deleted = front() //get the max ball
            ball_heap[1].set_state_deleted() //change the state
            ball_heap[1] = ball_heap[size]
            size -= 1
            heapify_down(1, priority)             → O(?)
            max_addition += max_deleted
            player_turn += 1
        else
            ball_heap[1] = ball_heap[size]
            size -= 1
            heapify_down(1, priority)             → O(?)
    return max_addition
```

$all\ minor\ operations \rightarrow O(1)$

Hence, in this case, time efficiency of the heapify down method needs to be identified. If we assume a constant time 1 $(or\ c)$ for 1) a comparison to identify the priority ("HEADS" or "TAILS") and 2) a comparison to identify whether the current top node has a smaller value than its child nodes, 3) swapping operation, and 4) all other assignment and division operations, each heapify down method in the worst-case takes $\log_2 n$.

```
Algorithm heapify_down(c_index, priority)
  while (c_index * 2) <= size of the queue do          → O(log2 n)
    max_child_index = get_max_child(c_index, priority)
    if priority="HEADS" //Scott's turn
      if ball_heap[c_index].get_value() < ball_heap[max_child_index].get_value()
        swap(c_index, max_child_index)
    if priority="TAILS" //Rusty's turn                                  O(1)
      if ball_heap[c_index].get_digit() < ball_heap[max_child_index].get_digit()
        swap(c_index, max_child_index)
      if ball_heap[c_index].get_digit() == ball_heap[max_child_index].get_digit()
        if ball_heap[c_index].get_value() < ball_heap[max_child_index].get_value()
    c_index = max_child_index
```

Therefore, putting all this together for the delete operation,

$$Play\ rounds: \frac{n}{t} * extracting\ and\ deleting \max balls: 2t * heapify\ down: \log_2 n$$

$$= \frac{n}{t} * 2\mathrm{t} * \log_2 n = 2n \log_2 n$$

Finally, adding building two queues and extracting and deleting max balls together:

$$Building\ two\ queues: 2\mathrm{n}\log_2 n + Playing\ rounds: 2\mathrm{n}\log_2 n = 4\mathrm{n}\log_2 n \leq c * \mathrm{n}\log_2 n$$

Therefore, ignoring a constant c, this algorithm's upper bound time efficiency is $O(\mathrm{n}\log_2 n)$.

The lower bound time efficiency occurs where the list of balls is sorted. In this case, building two queues takes $2n$ without the heapify up and playing rounds still takes $n \log_2 n$ where the balls do not have to be removed. This scenario, hence, still gives $\Omega(\mathrm{n}\log_2 n)$ for the best case.

Space efficiency is also derived through examining the main methods, but unlike time efficiency, space efficiency is simpler because most operations operate, such as swapping, within the existing space.

When there are n balls, building two queues require $2n$ space. Playing rounds occur within the same queues, therefore, ignoring individual temporary variables (such as a temp variable in the swap method or max addition variable in the delete method), playing rounds uses existing $2n$ space. The results of total scores are stored in a dictionary with keys so this requires a constant $1\ (or\ c)$ space. Space efficiency of $o(n)$ does not change with the best or worst-case scenario because the algorithm essentially requires two heaps to be built.

$$Building\ two\ queues: 2n\ and\ Playing\ rounds: 2n\ (existing\ queues)$$
$$+ Storing\ the\ result: 1 = \ 2n + 1 \leq c * \ n$$

Conclusion

In this assignment, the algorithm to maximise the sum of the selected balls is discussed. By using priority queue based on a heap, extracting the maximum value of balls displays time efficiency of $\theta(\mathrm{n}\log_2 n)$ while space efficiency is bound to $\theta(n)$ since most operations utilise in-place swapping and individual temporary variables.