# 2801ICT: Assignment 2

"Maximize the Score" Report

## 1 Table of Contents



## 2 Overview

This report documents the design and implementation of an algorithm to solve the "*Maximise the Score*" problem in which two players, Scott and Rusty, are competing to beat one another. Given a *n* number of balls each with a certain value associated with it, Scott and Rusty want to choose balls from the table so that they can get the highest over all score possible. They are given *k* number of turns over *T* number of games. The player to choose first is decided by a coin toss where Scott will start if the coin lands on *heads* and *tails* means Rusty goes first.

The two players have different strategies on choosing the optimal ball in each turn. Scott will choose the ball on the table that has the maximum raw value. However, Rusty will choose the ball that has the greatest sum of digits from its value. For example, if the table had two balls on it, one with value 50 and another with value 24, Scott will choose 50 as:

$$50 > 24$$

But Rusty will choose 24 because the sum of those digits is the highest amongst the balls:

$$2 + 4 = 6 > 5 + 0 = 5$$

The total score for each player is the sum of the raw values for all of the balls they have chosen throughout the game. We will see how these approaches effect who wins each game, which is the most optimal. Each player will use a priority queue to decide what the best ball is. This queue will be represented as a maximum heap, which will work slightly different for each player based on their play styles. The rest of the report will cover the solution to this problem.

## 3 Algorithm Descriptions

This implementation uses two classes and a driver program. This section will cover their applications.

### 3.1 Ball Class

A very simple class, this is how the balls are represented. Each ball is an object created from this class and they all contain information on the balls raw value, the sum of its value's digits, as well as whether the ball has been chosen by a player or not. Along with this data, a few functions are provided that allow other parts functions through the program to operate on it and retrieve its information. *Figure 1* shows a graphical representation of a ball object with its raw value and sum of digits.

When a new ball object is created, the sum of the digits of its raw value is calculated and then stored. Initially, the ball is recorded as not chosen by a player. Both the raw value and sum of digits use ***unsigned long long*** data types to allow for extraordinarily large values.

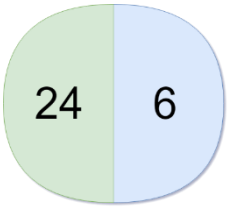


*Figure 1: A ball object.*

### 3.2 Heap Class

Each player is represented with a maximum heap application of a priority queue. Even though Scott and Rusty play differently, the same class is used for both with slight differences in the class's functions to account for the different play styles. In this way, both heaps will still prioritise the optimal ball based on the two playstyles and hence, *player*, *Scott*, or *Rusty* are synonymous with *heap*. Also, heaps use ball object references as nodes and therefore *ball* is synonymous with *node*. More on that later.

#### *3.2.1 Upheaping*

Both heaps are left complete which means that new nodes will be inserted from left to right. When a Scott-heap inserts

a ball, it first inserts the ball in the last available position in the heap. It then records the sum of digits for that ball, but when the ball is upheaped to be placed in the appropriate position, only the raw value is considered. Hence, a ball will keep moving up through its ancestors until it reaches a root node, or its current parent has a greater raw value than itself. *Figure 2* shows this process in action.

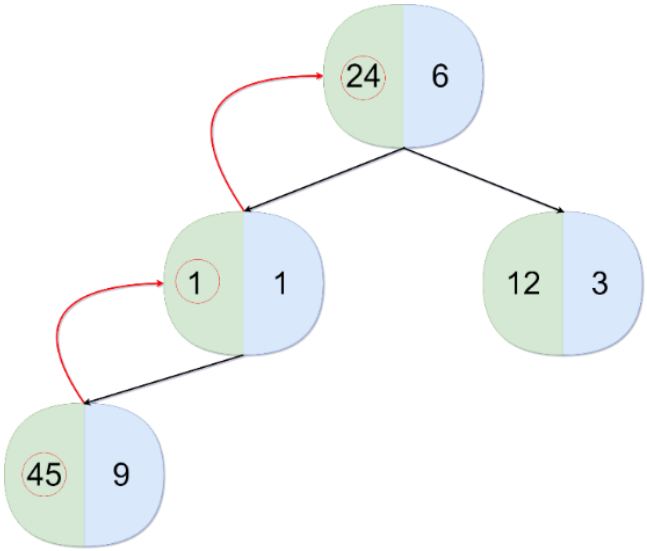


*Figure 2: Upheaping in a Scott-type heap.*

Upheaping is more complex for a Rusty-heap because there is more than one case that must be considered; as Rusty chooses his balls based on their sum of digits from their raw value, there can be situations in which Rusty will choose a ball that is not optimal. Assume Rusty has a choice between two balls, 24 and 42. One can see that the sum of their digits is the same:

$$2 + 4 = 6 = 4 + 2$$

But it is also clear that that 42 is a larger value than 24 and hence should take priority. A problem arises when Rusty will only upheap a ball based solely on it's sum of digits, this could lead to a circumstance in which a ball with an equal sum of digits to its parent but with a *greater* raw value than its parent won't take priority, causing Rusty to play sub optimally. Therefore, the three cases that must be considered when Rusty performs an upheap are:

1. The child node's sum of digits is greater than its parent's sum of digits. The simple case in which a swap is performed.
2. The child has an equal sum of digits to its parent, but a smaller raw value. In which case, no swap will take place.
3. Both the child and its parent have the same value for their sum of digits, but the child has a *greater* raw value. If this happens, swap them.

*Figure 3* shows a technically correct heap that doesn't account for case 3, which would cause sub-optimal selection. Therefore, Rusty must do one more check than Scott. When he encounters a parent and child with the same sum of digits, he then must check the raw values for both. If the child's raw value is greater than he will swap them.

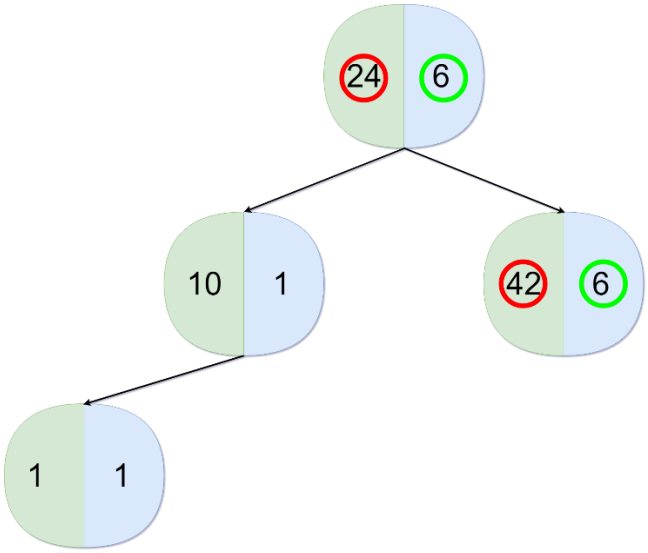


*Figure 3: A technically correct, but sub-optimal heap.*

### 3.2.2 Retrieving the Maximum Node

If one has correctly implemented node insertion using upheaping, then the node with the maximum/most optimal value will be at the root. Hence, "choosing" a ball is done by simply retrieving the root node of a heap. If there is only one node in the entire tree, i.e. only one ball to choose from, then that node will be returned and that's it for the operation. However, with more than one node in the tree, extra steps must be taken to ensure the next root node will be the optimal one.

In which case, the root node will be removed and returned, the heap will place the last node in the tree as the root, and finally will perform a downheap operation to ensure that node is in its correct place and the root has the new maximum.

### 3.2.3 Downheaping

For both a Rusty-type heap and a Scott-type heap, this operation works very similar to upheaping. The only difference is it is going down the heap tree. Also, as with upheaping, the process is simpler for Scott than it is for Rusty. *Figure 4* shows how Scott performs this process.

The parent node will check which of its children has the greatest value (if they do at all). If so, then the parent will be swapped with the child and so on until the node being operated on reaches the end of the tree or neither of its children have greater values.

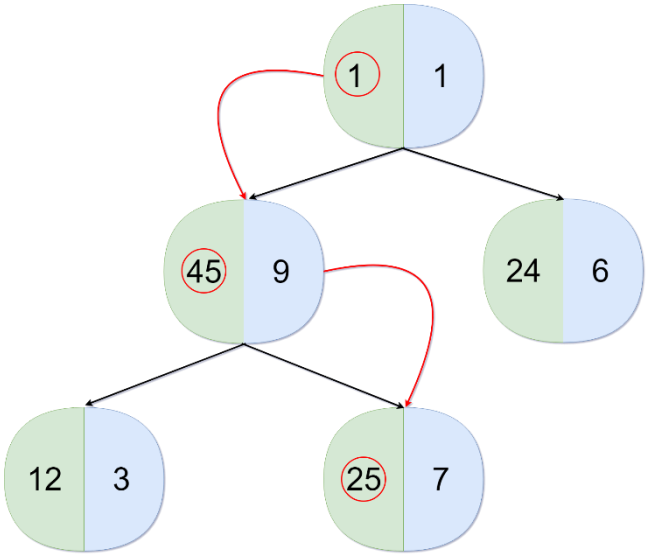


*Figure 4: How Scott will swap nodes when downheaping.*

As discussed earlier, Rusty must take extra steps to ensure his heap is always in an optimal state. The procedure of downheaping is much the same for him as is upheaping. The same three cases must be checked, but instead of a child node assessing if it has a higher priority than its parent, this time it is the parent assessing which of its children has a higher priority than itself. This continues until the node people operated on reaches the end of the tree or neither of its children have greater values than itself.

## 3.3 The Driver Program

This is the main function that drives the problem solution with heaps and ball objects. As shown previously, each player (Scott and Rusty) is represented by a maximum heap priority queue. These heaps don't contain ball objects themselves; they use references to these balls. The objects are contained in a ball class array in the driver program. In this way, if a ball is chosen by a player it is recorded as having been picked (it doesn't matter by who) and the reference to that object in the other player's heap will show as the same. This approach was used in favour of each player having a separate collection of ball objects so that when one player removes all ball, the other does not have to search through his heap, find the ball, then remove it. The downside to this approach is that a player will only know if a ball has been chosen if he attempts to choose it himself. When that happens, he will pop the ball reference, discard the ball, get the next highest priority ball, and so on until he finds one that has not been chosen.

The main loop of the game is simple; if there are balls to be chosen from the table, let the first player (decided by the coin toss) choose up to *k* balls. Then let the next player choose *k* balls and so on. If a ball has not been chosen and a player selects it, the raw value of that ball is then added to the player's total score.

Finally, the scores are displayed for both players.

# 4 Algorithm Pseudocodes

## 4.1 Ball

### *4.1.1 Calculating the Sum of Digits.*

```
ALGORITHM CalculateSumOfDigits():
        //Calculates the sum of digits for a ball's raw value.
        sum ← 0
        actualValue ← thisObj.value
        while actualValue ≠ 0 do:
                sum ← sum + actualValue % 10
                actualValue ← actualValue / 10
        end while
        thisObj.sumOfDigits ←sum
```

## 4.2 Heap

### *4.2.1 Inserting a Node.*

#### *4.2.1.1 Rusty Version*

```
ALGORITHM InsertNode(Ball *ball):
//Input: A ball object reference.
//Inserts a ball into the heap in its correct place.
thisObj.Add(ball)
thisObj.currenHeapSize ←thisObj.currenHeapSize + 1
currentNodeIndex ←thisObj.currenHeapSize - 1
parentSumOfDigits ←thisObj.parent.sumOfDigits
currSumOfDigits ← thisObj.sumOfDigits
while currentNodeIndex ≠ 0 and parentSumOfDigits ≤ currSumOfDigits:
        if parentSumOfDigits = currSumOfDigits:
                parentValue ← thisObj.parent.value
                currentValue ←thisObj.value
                if parentValue < currentValue:
                        swap thisObj.parent and thisObj
                        currentNodeIndex←thisObj.parent.index
                        parentSumOfDigits ←thisObj.parent.sumOfDigits
                        currSumOfDigits ← thisObj.sumOfDigits
                end if
        else if parentSumOfDigits < currSumOfDigits:
                swap thisObj.parent and thisObj
                currentNodeIndex←thisObj.parent.index
                parentSumOfDigits ←thisObj.parent.sumOfDigits
        currSumOfDigits ← thisObj.sumOfDigits
        end if
end while
```

#### 4.2.1.2 Scott Version

```
ALGORITHM InsertNode(Ball *ball):
//Input: A ball object reference.
//Inserts a ball into the heap in its correct place.
thisObj.Add(ball)
thisObj.currenHeapSize ←thisObj.currenHeapSize + 1
currentNodeIndex ←thisObj.currenHeapSize - 1
parentSumOfDigits ←thisObj.parent.sumOfDigits
currSumOfDigits ← thisObj.sumOfDigits
while currentNodeIndex ≠ 0 and parentSumOfDigits < currSumOfDigits:
        swap thisObj.parent and thisObj
        currentNodeIndex←thisObj.parent.index
        parentSumOfDigits ←thisObj.parent.sumOfDigits
        currSumOfDigits ← thisObj.sumOfDigits
```

### 4.2.2 Removing the Maximum Node

```
ALGORITHM RemoveMaxNode()
//Output: A ball object pointer.
//Retrieves the ball at the root node. Then performs a downheap procedure.
if thisObj.heapSize = 1:
        thisObj.heapSize ←thisObj.heapSize – 1
        return thisObj.rootNode
else
        returnVal ←thisObj.rootNode
        thisObj.heapSize ←thisObj.heapSize – 1
        thisObj.rootNode ←thisObj.lastNode
        thisObj.Heapify(0)
        return returnVal
end if
```

### 4.2.3 Downheaping.

#### 4.2.3.1 Rusty Version

```
ALGORITHM Heapify(int n):
leftChildIndex ←thisObj.GetLeftChild(n)
rightChildIndex ← thisObj.GetRightChild(n)
currObj ← thisObj.heapArray[n]
largestChildIndex ←inputIndex
if leftChildIndex ≤ thisObj.heapSize
and thisObj.leftChild.sumOfDigits ≥ currObj.sumOfDigits:
        if thisObj.leftChild.sumOfDigits = currObj.sumOfDigits
        and thisObj.leftChild.value > currObj.value:
                largestChildIndex ← leftChildIndex
        else if thisObj.leftChild.sumOfDigits > currObj.sumOfDigits:
                largestChildIndex ← leftChildIndex
        end if
end if
if rightChildIndex ≤ thisObj.heapSize
and thisObj. rightChild.sumOfDigits ≥ currObj.sumOfDigits:
        if thisObj.rightChild.sumOfDigits = currObj.sumOfDigits
        and thisObj.rightChild.value > currObj.value:
                largestChildIndex ←rightChildIndex
        else if thisObj.rightChild.sumOfDigits > currObj.sumOfDigits:
                largestChildIndex ←rightChildIndex
        end if
end if
if largestChildIndex ≠ inputIndex:
        swap currObj and thisObj.heapArray[largestChildIndex]
        thisObj.Heapify(largestChildIndex)
```

#### 4.2.3.2 Scott Version

```
ALGORITHM Heapify(int n):
leftChildIndex ←thisObj.GetLeftChild(n)
rightChildIndex ← thisObj.GetRightChild(n)
currObj ← thisObj.heapArray[n]
largestChildIndex ←inputIndex
if leftChildIndex ≤ thisObj.heapSize
and thisObj.leftChild.value > currObj.value:
        largestChildIndex ←leftChildIndex
end if
if rightChildIndex ≤ thisObj.heapSize
and thisObj. rightChild.sumOfDigits > currObj.sumOfDigits:
        largestChildIndex ←rightChildIndex
end if
if largestChildIndex ≠ inputIndex:
        swap currObj and
thisObj.heapArray[largestChildIndex]
        thisObj.Heapify(largestChildIndex)
end if
```

### 4.3 Main Driver Function Loop

**ALGORITHM** MainLoop(n, k):
//Input: n is the number of balls.
: k is the number of turns.
// Plays the "Maximize the Score" game.
**while** *n* > 0:
**for** *i* ←0 **and if** *i* < *k* **and** *n* > 0:
*tmp* ←*player1.MaxNode*()
**if** *tmp.notChosen*:
*player1.UpdateScore*(*tmp*)
*tmp.SetChosen*(**true**)
*n* ← *n* – 1
*i* ←*i* + 1
**end if**
**end for**
**for** *i* ←0 **and if** *i* < *k* **and** *n* > 0:
*tmp* ←*player2.MaxNode*()
**if** *tmp.notChosen*:
*player2.UpdateScore*(*tmp*)
tmp.*SetChosen*(**true**)
*n* ← *n* – 1
*i* ←*i* + 1
**end if**
**end for**
**end while**

## 5 Algorithm Analysis

It is well known that the best-case complexity for retrieving the max node from a heap is *O(1)* and this holds true for my heap implementation, assuming there is only one node. With the assumption that there are no nodes in the heap, the best-case complexity for inserting a node is also *O(1)*. But what about the worst-case? It is also known that the worst-case efficiency for both inserting a node and heapifying after retrieving the maximum node is $O(\log_2 n)$.

To test my heap, I generated random 100 random balls with increasingly large values so that every time a ball was inserted, it would perform the worse case efficiency. *Figure 5* shows the resulting number of operations executed versus $O(\log_2 n) + 1$ .

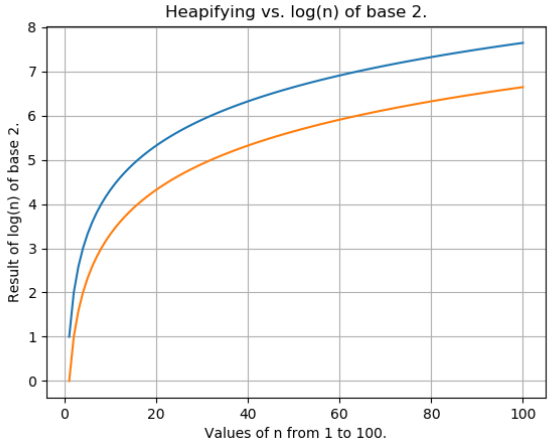


*Figure 5: Showing how my algorithm (orange) performed against log(n) + 1 (blue).*

Conversely, I generated 100 more test balls so that every time one was removed from the heap, it would have to do the worst-case time complexity. The result was the same as in *Figure 5*.

Because the game has two players and *n* number of balls, the amount of operations executed in the game's main loop will happen *n* amount of times for each player and the heapifying of the players' heaps will be $O(\log_2 n)$ which has been shown in the worst case scenario. Hence, the amount of executions is:

$$O(n \log_2 n)$$

As before, I created 100 test balls that to test my algorithm and hypothesis for the worst-cast time efficiency. As can be seen by *Figure 6*, this is indeed it's worst-case time complexity.

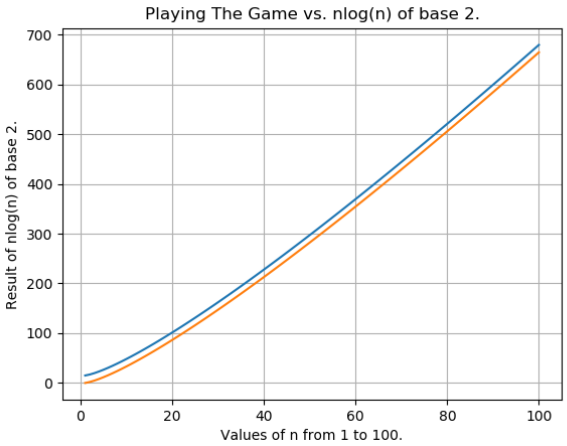


*Figure 6: Showing how my game (orange) performed against nlog(n) + 15 (blue).*

# 6 Results

| **Scott** | **Rusty** |
|---|---|
| 1000 | 197 |
| 240 | 150 |
| 2100000000 | 98888899 |
| 9538 | 2256 |
| 30031 | 17796 |
| 4726793900 | 3941702128 |
| 13793 | 12543 |
| 2173 | 1665 |
| 3923529875 | 3049188235 |
| 0 | 284401 |

With the exception of the very last case, it is clear that Scott's strategy is the most optimal one. In the last case, Rusty got to pick first and the amount of turns was equal to the amount of balls. So, he chose them all and the game was over.

Lastly, I omitted how fast each game took because no matter what I used to get the most accurate reading possible, the game time was always too small to even calculate and would result in a time of 0.0.