Computing Algorithms

ASSIGNMENT 2 REPORT

Himath Ratnayake
S5209861 | GRIFFITH UNIVERSITY 2021

Algorithmic Design

Task Overview:

This problem details a game known as "Maximise The Score" between two individuals; Scott and Rusty. In it, there is a table with an "n" number of balls, and their goal is to pick balls from the table, such that they can maximise their score.

In each game, a coin is tossed. If the coin is heads, then Scott will start the first round. If it is tails, Rusty will start the first round; they will alternate rounds from there until the number of balls placed on the table runs out.

At each round, the current player has a "k" number of turns where they can pick balls such that they can maximise their score. However, for Rusty, he will only pick balls whose digits will sum to the greatest value. For example, for the numbers 51 and 102, 5+1=6 and 1+0+2 = 3, so 51 will be chosen by Rusty as it is a "larger" number by digit sum. Scott however does not have any constraints on the ball he wishes to pick, except the fact that both players are assumed to always be playing optimally. Therefore, Scott will always try to pick a ball with a value that will maximise his current score.

In the problem, a text file is given with a variety of inputs. The first line in the text file denotes the number of test cases there are. This means that there is 10 different test cases that must be accounted for going by the input file contents. For each test case "T" from 1 to 10, the next 3 lines denote the specifics of said test case. For example, line 0 would be the number "T", then line 1,2,3 would be for test case 1, then line 4,5,6 for test case 2, and so on. In the first line for the test case, the number of balls on the table (n) and the number of turns that can be taken by each player per round (k) is denoted. The line after shows the values of the "n" balls. The third line for each test case will be a string of either "HEADS" or "TAILS", which will denote which player will start the game.

The goal of this problem is to simulate this game, assuming both players play optimally, such that we can find their final scores for each test case.

Algorithm Description:

First, each of the inputs for test cases is read in through the process_inputs function. Given a test case number "i", the next 3*(i-1) + 1 to (3*i) + 1 lines will contain the relevant lines for the given test case "i". The lines between these indexes in the document are saved in a "testlines" list. The first index of the first line in the testlines list is the number of balls, and the second index in the first line is the max amount of turns. Then, the second line is the ball score values and the third line denotes the toss result.

Then, the general idea for this solution is two use two priority queues – one for Scott and one for Rusty. A priority queue is a type of data structure where every element is associated with a key, and the queue returns the element according to this key. This assignment uses this strategy with a max-priority queue using binary heap structure, which will return the element with the highest key first. For the constraints of the problem this is ideal as we are looking to "maximise" the score, as explained in the algorithm overview. In a priority queue with heap, the largest element will always be the element at the tree's root. The algorithm must also maintain heap order property, which is to say that the parent of the child nodes will always have a greater than or equal key value than the children.

For Rusty:
Each element of his queue will consist of a pair of numbers in a list. The first number in this pair will be the ball score, as read in from the input file. The second number will be the sum of that given ball's score.
For Scott:
Each element of his queue will also consist of a pair of numbers in a list, but this time the element order within the pair is swapped; the first number will be the sum of a given ball's score, and the second number in this pair will be the ball score.

Both lists are then passed into a heapify function, which will arrange the queues of Scott and Rusty in a binary heap order. First the largest element is initialised as the root. Then, each of the child nodes (either the left or right) is checked to see if a value is present which is greater than the value of the root.

All left children can be found using the pattern of (2i +1) where i is the current index and parent node. Similarly, all right children can be found with the pattern of (2i +2).

If such a value exists, then the root is swapped such that the larger value is the new root. This algorithm is recursively run until the case that the parent node is greater than both child nodes returns as true, which is to say that the max-heap property has been restored.

After this, in a while loop Scott and Rusty's max queue items are popped and added to their scores. When the toss result is heads, Scott's queue's highest priority (maximum) element will be added to Scott's score. Then, the index of this element will be found in Rusty's queue and popped. It will also be popped from Scott's queue. This is so that it can't be reused in either queue again. Then, the toss result changes to the opposite of the current (if its currently Heads, it will change to Tails, and vice versa) to signify a new turn. This algorithm works because the priority elements in each of the two queues is different – in Rusty's queue we're sorting in order of priority determined by the sum of the balls, so the balls with greatest sum will be popped first as that is what his priority is in the scenario. Conversely, in Scott's queue we're sorting in order of highest value element, as he doesn't care about the sum of balls.

Algorithm Pseudocode:

Main Algorithm Pseudocode For Each Test Case:

```
Initialise a heap queue for both Scott and Rusty

Initialise scores for both Scott and Rusty, starting at 0

Create a heap of pairs for Rusty’s queue based on the value of all the balls. Of the pair, the first value
is the actual ball value and the other is the sum of ball values

Then, create Scott’s heap, where the sum of ball values will appear first, then the actual ball values

Then, run the heapify on both Scott and Rusty’s queues to build them

In a while loop, while neither of the priority queues are empty:

    If the coin flip is on Heads, Scott will start, with a for loop until the round’s max turns:

        Add the maximum value on Scott’s heap to Scott’s score

        Then, find the index of that same value in Rusty’s heap and pop it so it can’t be used
        again

        Finally, run popq on the maximum value from Scott’s heap as it won’t be used again

    At the end of the for loop, the coin flip string becomes Tails to signify a new turn

    If the coin flip is on Tails, then it’s Rusty’s turn, with another for loop that will iterate until
    the maximum turns value for this test case:

        Find the index of the maximum value of Rusty’s queue in Scott’s queue

        Add Rusty’s maximum value to Rusty’s score

        Run Popq the maximum value of Rusty’s queue from Scott’s queue

        Then, pop the max value from Rusty’s queue as well with popq

    Switch the coin flip string back to Heads to signify a new turn – continue this process until
    both queues are empty, then return Scott and Rusty’s scores
```

Pop Operation Pseudocode:

```
Swap the value of the ball at index “i” with the value of the ball at the end of the queue
Then, pop the last element with the .pop() function, which was the originally the value at index i
Call the heapify function to restore the binary max heap property
Return the ball_scores array back to the main algorithm
```

Heapify Operation Pseudocode:
The index “i” passed in is set as the largest value. The left child is at index (2i+1) and the right child is at index (2i+2),
If the left child exists (its size is less than the size of num_balls value passed in), then the run the following checks:

- If the 2nd element in the list present at the index “leftchild” is greater than the second element of the list at index “largest”
- OR if the 2nd element in the list present at the index “leftchild” is equal to the second element of the list at index “largest” AND the first element of the list at index “leftchild” is greater than the first element of the list at index “largest”
    - If either are true, the largest index is now equal to the left child index

The same checks are repeated for the right child.
Then, if the largest does not equal the original “i”, which is to say the largest index has been changed after running the above checks:

- Swap the the values of index i and index "largest" as i was found not to be the largest anymore
- Run heapify recursively again
- Continue running recursively until heap structure is restored, which is when largest will remain as the “i” initially set at the start of the function

Then, returned the new heapified list.

Results

For all inputs, a correct result was found in the python program (see Appendix for results shown within the actual program terminal – for easier reading these results were converted into tabular format, and output.txt when running on my computer has been attached to the zip file submission)

Input	Output	Correct Answer?	CPU Time (Secs)
1	1000 197	Yes	0.000179
2	240 150	Yes	0.000432
3	2100000000 98888899	Yes	0.000179
4	9538 2256	Yes	0.000511
5	30031 17796	Yes	0.003272
6	4726793900 3941702128	Yes	0.000552
7	13793 12543	Yes	0.001320
8	2173 1665	Yes	0.003837
9	3923529875 3049188235	Yes	0.000627
10	0 284401	Yes	0.001813

Table 1: Results from running the code

Algorithm Correctness

There are various ways to prove the correctness of this algorithm.

One such way is through empirical analysis – that is to say, the algorithm is run with various inputs and the given output from the code is compared with the actual theoretical output. As can be seen by table 1, the algorithm was able to calculate and give correct scores for both Rusty and Scores in all ten test instances. The scores were not approximate to the output – they were exactly the same, alluding that for most if not all possible test cases, the exact correct output will be given. However, this analysis is often not enough as there can be countless more unexpected test inputs that we potentially may have not accounted for. Therefore, the algorithm's process can be formally reasoned.

The actual pattern in which the scores is distributed is based on the values read in from file, and with the given pseudocode in the section above, it will always follow a pattern of:

- One person starts and does "max_turns" amount of turns where items from their queue are added to the score and then popped
- Then, the other person does "max_turns" amount of turns as well, with the same process
- This process continues until both people's queues are empty.

This pattern is correct given the constraints of the task, and can't really be changed in a way that will provide a wrong answer so long as the score items taken from the queue are popped in the correct order. Therefore, arguably the most important aspect of this algorithm is the heapify function, as this is what dictates what order of scores that Rusty and Scott will actually end up with, and actually has potential to have failures for certain inputs if heapify is not implemented completely correctly.

Proof: Correctness When Building The Heap

Pseudocode:

```
Heap_Size for Queue = Length(Queue)
For i from Length(Queue)/2 to 1:
    Run Heapify
```

If the building of the max heap worked correctly, then a correct sorting through heapify must take place. The property we must maintain through the algorithm is heap[i] >= heap[2*i]. That is to say that each node in a tree has a key which is less than or equal to the key of its parent. Hence, if the max heap property was started off with and maintained throughout the algorithm, the algorithm must then be correct – the maintenance of this max heap property throughout each iteration is known as a loop invariant, and can be used to prove correctness.

The invariance in this property for each loop can be visualised like so:

1. **Initialisation Phase**: Before the first iteration of this pseudocode, every node (i+1, i+2,… n), is the root of a max-heap.
2. **Maintenance Phase**: At each loop, this max heap property is maintained: the children of node i are numbered higher than i. Hence, by the loop invariant, they are both roots of max-heaps. The heapify function preserves this idea too as seen in the code, where (i+1, i+2,… n) are all roots of max-heaps
3. **Termination**: when i reaches 0, each node is now the root of a max heap

Hence, the building of this heap is proved to be done correctly.

Next, for the actual heapify function, proof by induction can be done. When doing induction, it is useful to know the problem size, post and pre-conditions, as well as define a clear base case and inductive step. For max heapify, which is the type of heapify being done in this algorithm:

- **Precondition:** trees rooted to the left and right are heaps
- **Postcondition:** tree rooted at i is a heap
- **Problem Size** = height of a node i
- **Base Case:** we must prove that heapify is correct for every input with height(i) = 0
- **Inductive Step**: A and i can be any input parameters that satisfy the precondition.
    - If this heapify algorithm is correct when the problem size is "k", we must then prove that it is correct when the problem size is "k+1"

Through the code, it can be seen that the precondition is always satisfied before the procedure for heapify is recursively called again. Values are always exchanged such that this property holds (refer to pseudocode for heapify above – figure 1 below illustrates this).

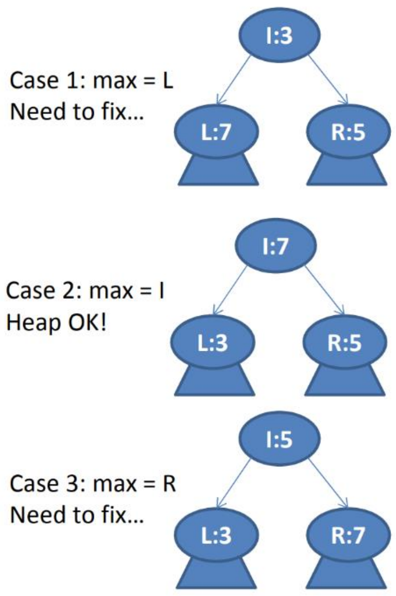


Figure 1: How a max heap will treat various nodes so that max heap property is maintained and preconditions are satisfied

Therefore, this algorithm can be confidently proven as correct for inputs as the building of the heap and the actual heapify function are operating as theoretically expected.

Performance Analysis

In this algorithm, there are 3 for loops which are run before that are responsible for preparing the priority queues of Scott and Rusty.

1. The first for loop is used to create the sums array, which stores the sum of digits for each value in ball_scores. Every mathematical operation within this for loop is done in constant(O(1)) time
2. The 2nd for loop is used to append ball scores and values from the sums array into both Rusty and Scott's queues. Appending to both "RustyQ" and "ScottQ" lists is done in constant O(1) time.

As both operations in the for loops above run at constant time, the run time is dependent on how many "n" number of times the for loop is done, giving a time complexity of O(n).

Next, a for loop is run for the heapify function to arrange both queues into a binary heap structure. The for loop itself runs at O(n) time. The heapify function's time complexity is O(log(n)), but building the heap initially can be done in O(n) time. Queue[n/2+1] to queue[n] are leaf nodes, and each node is a 1 element heap. The heapify function can be used in a bottom up manner which is more efficient. With this in mind, it can actually be seen that the time complexity is O(n) for building the heap. Please see the next page for the proofs on both building the heap in O(n) time, and the O(log(n)) time complexity proof for the actual heapify function.

Next, in the while loop, there are two possible options (toss result = heads or tails), and both have comparable time complexity. The Boolean expressions to see if toss result is heads or tails, as well as to check if both Rusty and Scott's queues are empty are run at O(1) constant time. The maximum element to add to the score is always at the root of the binary heap tree, so returning an array element at specific given index can be done in efficient constant time of O(1). However, there is also a for loop that traverses through Rusty or Scott's queue (depending on who's turn it is) to find the index of one person's maximum element in the other person's queue, which is done in O(n) time where n is the full length of the other person's queue in worst case (though the loop usually breaks before this is the case). Within this O(n) loop, the bool to compare a value at an index in one person's queue to another is run at O(1) time, and if it returns true, the popq operation occurs and the loop breaks.
This popq operation swaps the ball_scores array element at index i with that of the last element so that the value index i is now the last element. This last element is then popped. Both these operations occur at O(1) time. Next, the heapify function is called once again.

Apart from operations involving heapify, all other operations in this program can be condensed down into either O(1) or O(n), with O(n) dominating O(1). This leaves the heapify function; in fact, many operations in this algorithm run at O(log(n)) worst case time, owing greatly to the reliance of the "heapify" function which is responsible for the maintenance of heap property, where the parent node's value is greater than that of the children's. This is crucial as without this the priorities in the queue can't be maintained and the problem cannot be solved efficiently. A proof of the heapify function's time complexity is found below.

Proof of Heapify O(log(n)) time complexity:

Apart from building the heap, when calling the heapify function for other operations such as when a node is popped from the tree, the worst time complexity is O(log(n)).

In worst case, from the root of the tree, we are traversing it to the bottom. For a binary tree of size "n", the height can be log(n) at most – that is to say that with twice the number of elements, tree depth only increases by 1, so in the worst case the heapify is called once on each level of the heap. Hence, there are O(h) swaps, where h = height, and height = O(log(n)).

Because of this, as stated earlier, the heapify operation's worst case time complexity is O(log(n)) where all nodes must be traversed. This heapify method could also be theoretically called an "n" number of times at various points in the program, especially when popping from the queue. Hence, the final time complexity would be O(nlog(n)).

Proof of O(n) time complexity for building the heap:

Although heapify takes O(log(n)) time as proven above, and is also called "n" number of times while building the heap, it does not need to take O(nlog(n)) time to be built. This is because the heapify function itself doesn't always take O(log(n)) time – it takes less time for the nodes closer to the root.

At height *h* = 1, it requires n/2 swaps. At height *h* = 2, it requires n/4 swaps. At height *h* = i, it takes $n/2^i$ swaps. Hence, as height *h* increases as we move up the tree, the for loop runs from the index of the last internal node (size of the queue/2) with the height equal to 1, to the index of the root (1) with the height of log(n). Hence, it can be seen that heapify takes a different time for each node, which is O(h).

Therefore, the total time taken for making the heap is also dependent on the number of nodes with height *h*. A heap of size n is at most ($n/2^{h+1}$) nodes, when height is *h* and O(h) is the worst case time which heapify takes for each node. This can be summarised in the equation below – note that n has been moved out of the fraction as its constant and h has been used in its place.

$$= O(n \times \sum_{h=0}^{logn} \frac{h}{2^{h+1}})$$

$h/2^{h+1}$ can then be further simplified to $h/(2*2^h)$, but as the lone 2 is constant, it can be dropped as the other values will dominate it in big-O notation

$$= O(n \times \sum_{h=0}^{logn} \frac{h}{2^h})$$

As the summation approaches larger and larger values (up to infinity), it is found that it converges at 2, so this summation can be replaced with 2.

$$= O(n \times \sum_{h=0}^{\infty} \frac{h}{2^h})$$

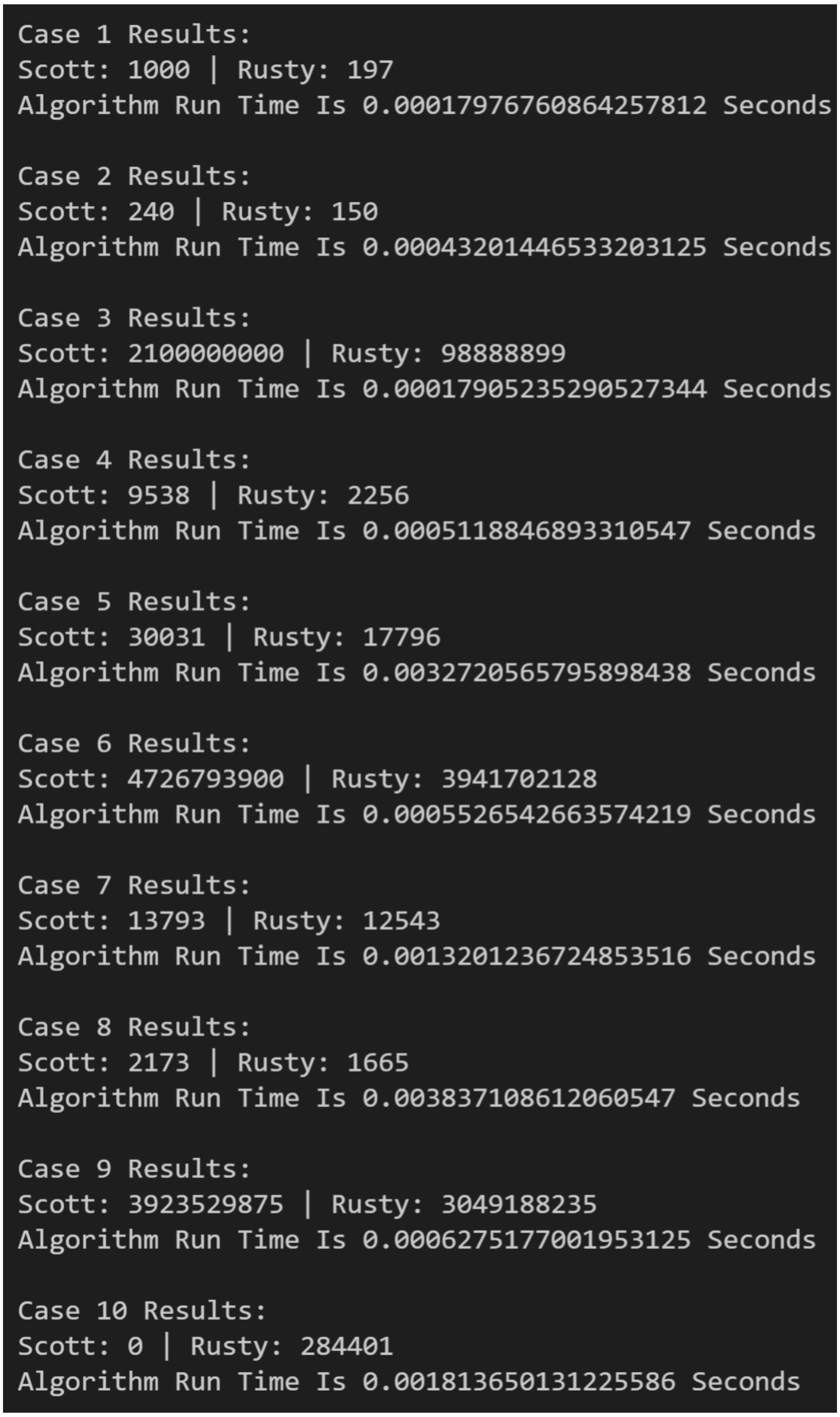
$$= O(n \times 2)$$

Once again, the 2 is a constant so it can be discarded. This gives a final time of **O(n)** for building the heap.

Final Time Complexity: ***O(n) + O(n*log(n))***

Thus, the final time complexity of the program can be deduced as ***O(n) + O(n*log(n))*** in the worst case, where **O(n)** is the time taken to build the heap and prepare the queues for Scott/Rusty, and **O(n*log(n))** being the time taken if the heapify operation were to run an "n" number of times at worst case in the algorithm.

Appendix



```
Case 1 Results:
Scott: 1000 | Rusty: 197
Algorithm Run Time Is 0.00017976760864257812 Seconds

Case 2 Results:
Scott: 240 | Rusty: 150
Algorithm Run Time Is 0.00043201446533203125 Seconds

Case 3 Results:
Scott: 2100000000 | Rusty: 98888899
Algorithm Run Time Is 0.00017905235290527344 Seconds

Case 4 Results:
Scott: 9538 | Rusty: 2256
Algorithm Run Time Is 0.0005118846893310547 Seconds

Case 5 Results:
Scott: 30031 | Rusty: 17796
Algorithm Run Time Is 0.0032720565795898438 Seconds

Case 6 Results:
Scott: 4726793900 | Rusty: 3941702128
Algorithm Run Time Is 0.0005526542663574219 Seconds

Case 7 Results:
Scott: 13793 | Rusty: 12543
Algorithm Run Time Is 0.0013201236724853516 Seconds

Case 8 Results:
Scott: 2173 | Rusty: 1665
Algorithm Run Time Is 0.003837108612060547 Seconds

Case 9 Results:
Scott: 3923529875 | Rusty: 3049188235
Algorithm Run Time Is 0.0006275177001953125 Seconds

Case 10 Results:
Scott: 0 | Rusty: 284401
Algorithm Run Time Is 0.001813650131225586 Seconds
```

Figure 2: Program results that are outputted to terminal when the program is run