Ball Picker Problem

Lucas Geurtjens | s5132841 | 12/05/2019

Table of Contents

Algorithm Design

Overview

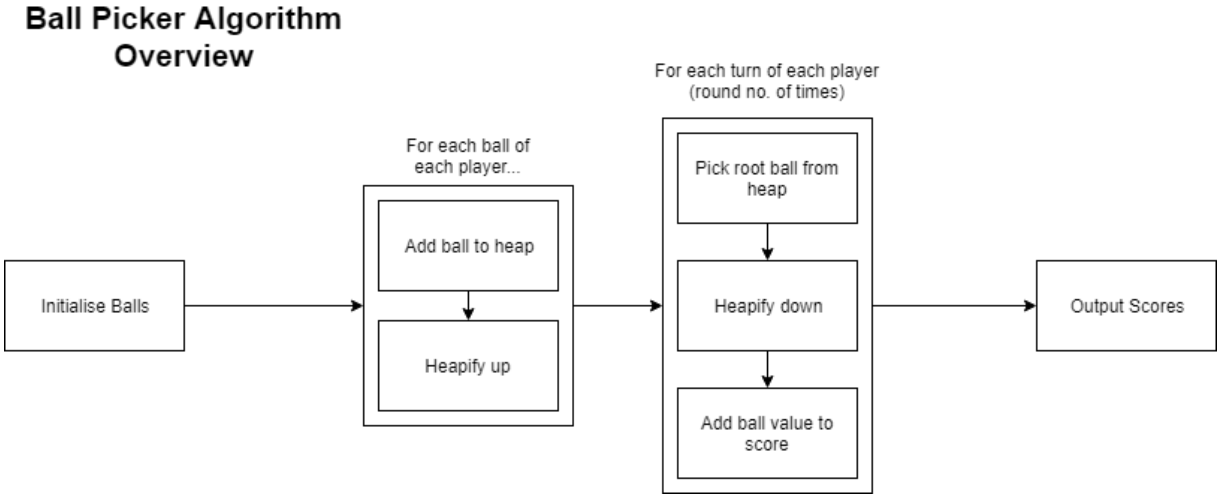

Figure 1: Ball Picker Algorithm Overview Diagram

The ball picker algorithm was able to optimally pick from a set of valued balls based on different player priorities. The algorithm followed the basic algorithm structure as seen in *Figure 1* and was implemented using priority queue heaps. Firstly, input regarding number of balls, turns per round and starting player are collected. A game is then begun. Within each game, a set of ball objects are initialised and shared across each player's heap. Each ball object has a status (whether the balls has been picked) and a list of two values for the players. Player Scott's value is the true value of the ball whereas player Rusty's value is the sum of the ball value's individual integers. From this, each player will add the balls into their heap array corresponding to their priority. Both players optimise their priority to find the largest value (within their individual values). This is done by calling a heapify up function each time a ball is added. With Rusty's valuing method, measures are taken to ensure he always picks the highest value ball if his two values are equal to each other by using the true value in those cases. With the ball heaps created, the player then takes turns picking balls based on whether a HEADS or TAILS was input. Each turn, the player is able to pick a certain amount of balls, popping the root node of their heap each time and performing a heapify down to resort the priority heap. To indicate the ball was taken, its status is changed to False. When a False ball is popped, the pop function is called again. After this, the true value of the ball is added to the player's score. This is continued until no balls are left, whereupon the scores of each player are output.

Algorithm Description

The Heap

The program utilised a heap data structure to maintain its priority queue with functions inspired by Gayle Laakmann McDowell [1]. Firstly, the heap uses several helper functions to perform its actions. These functions include getting indexes for, checking the existence of and retrieving the value of parent, left and right child elements in the heap. From this, a heapify up and heapify down function

are able to be created and used to pop and push elements from and on to the heap.

Heapify Up Example

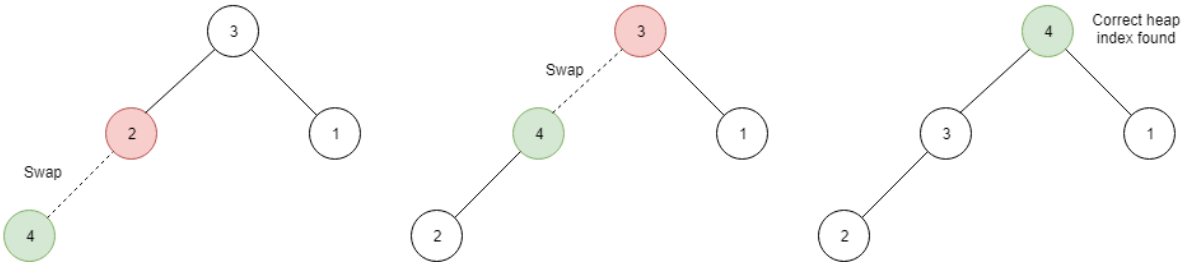

Figure 2: Heapify Up Example Diagram

As seen in *Figure 2*, the heapify up function is used to take a newly added node index (added in the last index), and compare it against its parent node, swapping the two if the index value is bigger than the parent. It should be noted that if the player is rusty and the index and parent values are equal, the algorithm will compare true values to decide which is bigger. Once an index has found a location where its parent is larger than itself (or has no parent index), it will secure its position in the heap.

Heapify Down Example

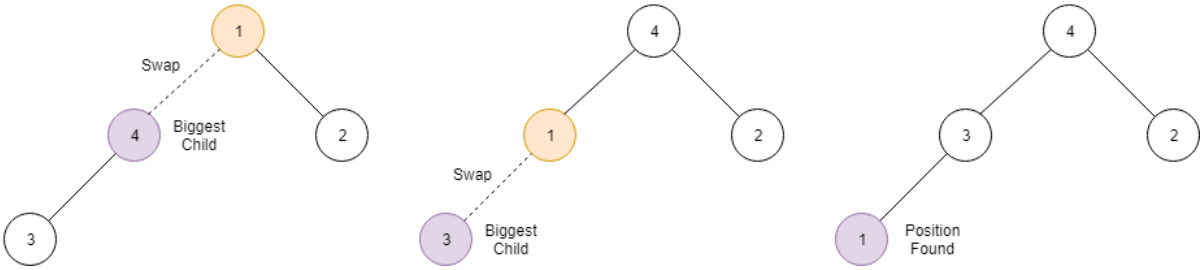

Figure 3: Heapify Down Example Diagram

Similarly, as seen in *Figure 3*, the heapify down function is used to take the root node (largest valued ball) and bring it to the right most leaf node position where it can be easily popped from the heap. Again, similar true value checking is used when values are equal for player Rusty.

Ball Initialisation

In order to create a set of balls to pick from, a list of ball objects was created. This was implemented as a ball class. Each ball object was given a status (whether the balls has been picked) and a list of two values for the players. Index 0 of the values list corresponded to player Scott's true largest value, whereas index 1 corresponded to player Rusty's integer sum value, which is calculated though a function that returns the sum of each number in a given integer from Scott's true value. These objects are initialised by taking the user input ball list and creating a list of objects with the value of the ball and status of True (unpicked). From this, the users are able to share this set of ball objects in their heap trees.

Game

In order to establish a game in which scores are calculated, a game function was used. Firstly, the current player is kept track of by a global player variable, which dictates which values from the objects should be used. This is updated every time the player changes. Next, once the ball objects are initialised, players create and add balls to their priority queue heaps. From this, the first player to begin picking balls is decided by the result of a coin toss in the user input (heads being player Scott

and tails being player Rusty). Next, while there still exists balls to pick, the chosen player is able to pick balls from the queue a specified amount of times. To indicate a ball was taken, its status is changed to False. If a False ball is picked, the pop function is called again and hence will disregard the ball. After this, the true value of the ball is added to the player's score. This is continued until no balls are left, whereupon the scores of each player are output.

Algorithm Pseudocode

```
# HEAP HELPER FUNCTIONS:

#  ========== 1 ==========
# Get index of left child
# Input: Index of parent node
# Output: Index of parent left child node

function get_left(parent_index)
return 2 * parent_index + 1


#  ========== 2 ==========
# Get index of right child
# Input: Index of parent node
# Output: Index of parent right child node

function get_left(parent_index)
return 2 * parent_index + 2


#  ========== 3 ==========
# Get index of parent node
# Input: Index of child node
# Output: Index of parent node

function get_left(parent_index)
return floor(child_index - 1) / 2)


#  ========== 4 ==========
# Check if an index has a left child node
# Input: A node index
# Output: True or False

function left_exists(index, array_size)

if get_left(index) < array_size
    return True
else
    return False


#  ========== 5 ==========
# Check if an index has a right child node
# Input: A node index
# Output: True or False

function right_exists(index, array_size)

if get_right(index) < array_size
    return True
else
    return False
```

```
#  ========== 6 ==========
# Check if an index has a parent node
# Input: A node index
# Output: True or False

function parent_exists(index)

if get_parent(index) < 0
    return True
else
    return False


#  ========== 7 ==========
# Get left child value
# Input: Index and reference to ball object
# Output: Value of index

function left(index, obj)
return obj[get_left(index)].value


#  ========== 8 ==========
# Get right child value
# Input: Index and reference to ball object
# Output: Value of index

function right(index, obj)
return obj[get_right(index)].value


#  ========== 9 ==========
# Get parent value
# Input: Index and reference to ball object
# Output: Value of index

function parent(index, obj)
return obj[get_parent(index)].value
```

Figure 4: Heap Helper Functions Pseudocode

```
# ============================
# Heapify Up
# Input: Reference to heap array
# Output: None

function heapify_up(heap_arr)

index = last element of heap array

while a parent node exists:

    if player is rusty and index value == parent value
        if true index value < true parent value
            break

    else if index value <= parent value
        break

    else
        swap(parent index, index)
        index = parent index

# ============================
# Heapify Down
# Input: Reference to heap array
# Output: None

function heapify_down(heap_arr)

index = 0 (root index)

while a child node exists:

    largest child = left child

    if player is rusty and left value == right value
        if true right value > true left value
            largest child = right child

    else if right value > left child
        largest child = right child

    if player is rusty and index value == largest child value
        if true index value > true largest child value
            break

    else if index value > largest child index value
        break

    else
        swap(index, largest child index)
        index = largest child
```

Figure 5: Heapify Up and Down Pseudocode

```
# ============================
# Add a ball to the heap
# Input: Ball object to add to heap
# and reference to heap array
# Output: None

function add_ball(ball_obj, heap_arr)

heap_arr.append(ball_obj)
heapify_up(ball_arr)


# ============================
# Pop a ball from the heap
# Input: Reference to the heap array
# Output: Root node

function pop_ball(heap_arr)

swap(first index, last index)
root = heap_arr.pop(last index)
heapify_down(heap_arr)

if root status is 'taken':
    root = pop_ball(heap_arr)

root status = 'taken'

return root
```

Figure 6: Push and Pop Ball Pseudocode

```
# ============================
# Initialise the set of ball objects
# Input: List of ball values
# Output: A list with ball objects

function init_balls(ball_values_list)

ball_object_list <- Empty list
default_ball_status <- 'Available' (True)

for ball_value in ball_values_list
    ball_object = Ball_Data_Type(ball_value, default_ball_status)

return ball_object_list


# ============================
# Set Rusty's value
# Input: True ball value
# Output: Sum of value's integers

function rusty_value(ball_value)

total = 0

ball_value_string = string_cast(ball_value)

for number in ball_value_string
    total += integer_cast(number)

return total
```

Figure 7: Ball Initialisation Pseudocode

```
# ============================
# Run a game of picking balls
# Input: Ball size, turns per round, ball list, starting player
# Output: Player scores

function game(ball_size, turns_per_round, ball_list, starting_player)

player <- global variable storing list [0, 0]
scott_balls <- Empty list
rusty_balls <- Empty list

ball_objects = init_balls(ball_list)

player = 0
for ball_index in ball_size
    add_ball(ball_objects[ball_index], scott_balls]

player = 1
for ball_index in ball_size
    add_ball(ball_objects[ball_index], rusty_balls]

rusty = starting_player

if True that rusty is starting player
    while balls_size is not 0

        player = 1
        for turn in range(turns_per_round)
            score[1] += pop_ball(rusty_balls).value
            ball_size--

        player = 0
        for turn in range(turns_per_round)
            score[0] += pop_ball(scott_balls).value
            ball_size--

else
    while balls_size is not 0

        player = 0
        for turn in range(turns_per_round)
            score[1] += pop_ball(scott_balls).value
            ball_size--

        player = 1
        for turn in range(turns_per_round)
            score[0] += pop_ball(rusty_balls).value
            ball_size--

return score
```

Figure 8: Game Pseudocode

Results and Algorithm Analysis

Testcase Results

Input #	Output (Scott | Rusty)	Time Taken (secs)
1	1000 197	< 0.000009
2	240 150	
3	2100000000 98888899	
4	9538 2256	
5	30031 17796	
6	4726793900 3941702128	
7	13793 12543	
8	2173 1665	
9	3923529875 3049188235	
10	0 284401	

Table 1: Test case results

As seen in *Table 1*, the algorithm was able to successfully produce results for the given input (as seen in appendix *Item 2*). The output solutions appear to be correct, as compared with a set of known solutions for the same test cases (as seen in appendix *Item 2*) and were each completed in a short period of time averaging below 0.000009 seconds indicating a potentially lower (non-exponential) time complexity.

Time Complexity

In order to find the time complexity of the algorithm, the algorithms various declarations and heap related functions were inspected. Firstly, two heaps were created for each player containing n ball objects. For each ball object, a $\log_2 n$ heapify up method was used. From this, during the game, there are n ball objects to be picked by the players, all of which require a heapify up method after being picked. Also, mixed throughout are low cost variable declarations and function calls made which can be all be rounded to a constant of 1.

Thus,

$$T(n) = n\log_2 n + n\log_2 n + n\log_2 n + 1$$

$$= 3n\log_2 n + 1$$

$$\leq C \cdot n\log_2 n$$

$$\therefore O(n\log_2 n)$$

Hence, the algorithm appears to have a time complexity of $O(n\log_2 n)$. This is a reasonably good time complexity and reaffirms the speed observations made in the the test case results.

Correctness

To check the correctness of the algorithm, we want to prove: When a ball is picked by one player, it cannot be picked by any other player. Here, proof by contradiction was used:

1. Assume this is not true. Then there must be a case where a ball can be picked by two players. Let's represent this ball as Ball' = Ball'' where both are the same ball object.

2. If Ball' is taken, it's status will be changed to False. When a ball is picked with a False status, it will be discarded, and the next Ball will be taken instead. Hence, it is impossible for Ball'' to be taken by the other player.
3. Hence, this is a contradiction and therefore the original statement is correct.

In-place and Stable

The priority queue heaping of the algorithm was investigated to see if it had characteristics of being in-place and stable. Firstly, it appeared that the algorithm was in-place. This is because no additional memory was used (apart from small data types such as integers) as manipulation of the list of balls only required a reference to the list. Furthermore, it appeared that the algorithm was not stable. This can be observed in *Figure 9*, in which the relative position of equal elements did not maintain its position. The cause for this can be seen in the heapify down function, where a newly added ball value is swapped unless it is larger than the child. Since the balls are equal, it is not larger, hence, will swap and change the relative order.

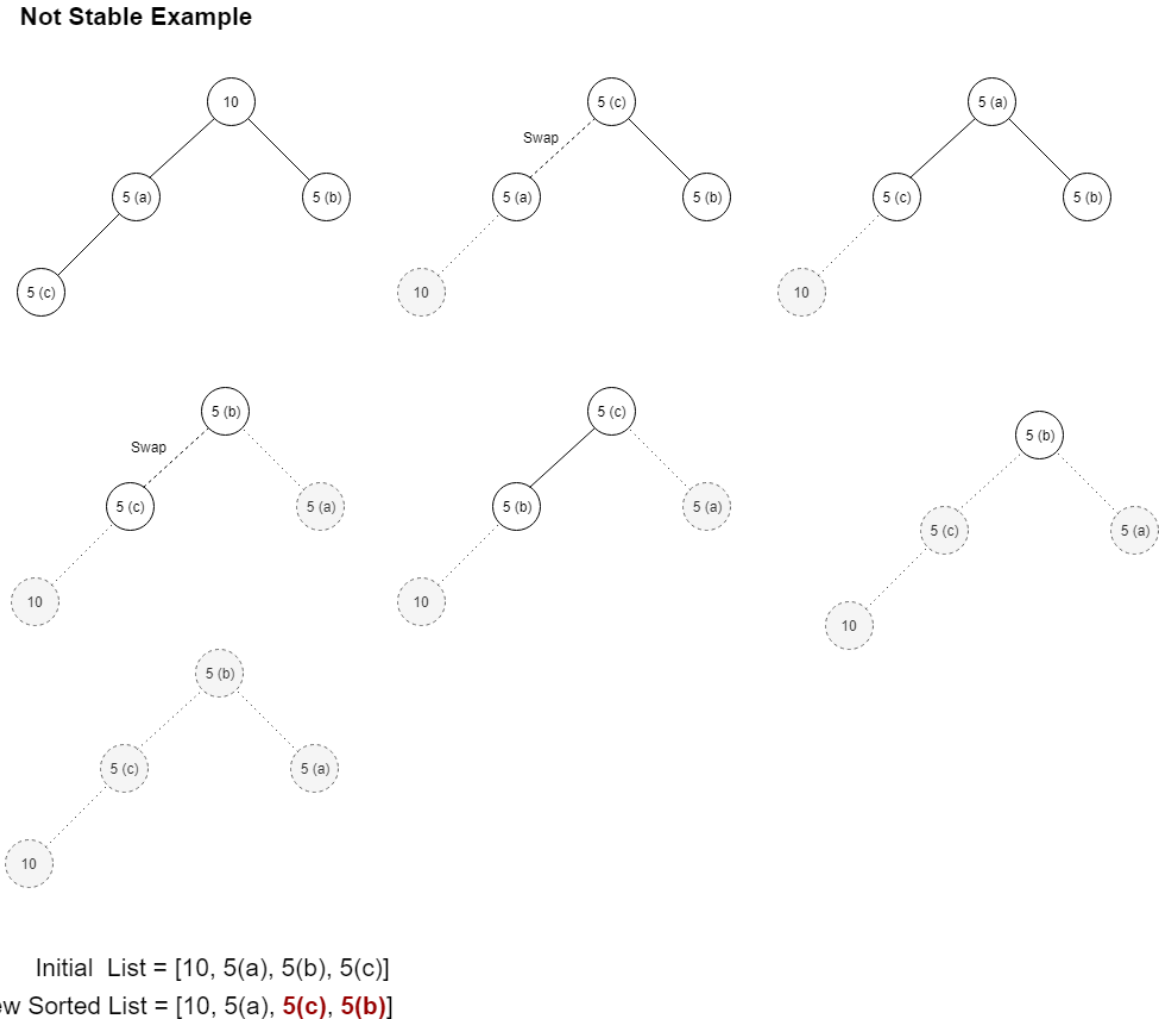

Figure 9: Example Demonstrating Algorithm is Not Stable

References

[1] Gayle Laakmann McDowell – *Understanding how heap data structures work* – 10/05/2019 – *https://www.youtube.com/watch?v=t0Cq6tVNRBA*

Appendix

Item 1: Known Input Solutions

Results for Python Implementation

Input	Output	CPU Time (Secs)	Input	Output	CPU Time (Secs)
1	1000 197	0.000029	6	4726793900 3941702128	0.000167
2	240 150	0.000051	7	13793 12543	0.000266
3	2100000000 98888899	0.000033	8	**2173 1665**	0.000827
4	9538 2256	0.000160	9	3923529875 3049188235	0.000119
5	30031 17796	0.001435	10	0 284401	0.000573

Item 2: Raw Input file used

```
10
3 2
1000 99 98
TAILS
8 2
50 60 20 10 90 30 60 70
HEADS
5 2
88888888 10000011 999999999 100000001 1000000000
HEADS
20 1
76 89 49 2 31 311 56 445 343 100 900 111 323 232 32 35 6456 565 760 878
TAILS
100 10
537 460 170 640 739 788 101 491 474 55 70 538 400 514 625 772 793 386 165 994 189
319 31 808 445 544 680 176 598 609 763 493 871 446 352 825 72 279 640 868 921 312
809 376 839 958 309 45 433 888 292 234 738 79 473 242 801 784 330 793 66 379 601
722 563 908 435 8 376 603 166 611 326 895 719 343 179 122 125 186 989 686 197 52
813 234 245 49 838 522 130 621 719 122 755 158 749 615 362 262
HEADS
20 5
937504347 151431905    672306410 431919895    681463104 131756378 206733105
709551656 193908628 630944659    169927916 365045527 483270672 832969883
667082001 309421734 378250713 13048925 157765172 544193398
TAILS
30 5
907 953 831 860 816 846 872 817 896 922 815 903 921 820 870 970 865 823 817 945 858
939 996 1000 909 810 945 861 754 795
HEADS
75 2
65 8 74 97 61 19 75 52 85 80 96 82 26 52 50 67 99 27 42 61 45 67 61 52 21 13 31 16
5 50 65 53 24 65 62 51 34 43 82 68 40 1 73 63 88 44 47 6 34 57 11 51 86 55 38 25 72
94 18 92 98 65 1 16 75 61 2 92 64 63 27 17 28 75 33
TAILS
15 2
995944177 13345781 233071387 718505424 365578597 290386139 796380040 72915505
362668340 329477805 256843388 208406422 680174492 924323770 724696843
TAILS
55 55
9951 9228 677 1051 6896 4156 7747 1820 5451 3453 1889 5729 8415 3593 1605 8288 2864
6748 8284 7679 1712 7326 8575 6743 6282 3300 2314 6740 8518 8919 4122 429 2959 1037
8073 4784 4550 4688 330 2337 4731 2231 8795 3620 5686 8473 9315 1045 5754 9633 5031
7286 6027 6786 726
TAILS
```