Advanced Algorithm

Dr. Moeini

1st Code Assignment

Parsa Mohammadpour

Hesam MumivandFard

Mohammad Rahbari Moghadam

Saeed Terik

Table of Contents

Median Maintenance

In the following, we provide the first question of the assignment and then we provide our solution for it and then we describe it. So, now let's move to the question.

Question

The goal of this question, is the implementation of '**Median Maintenance**' Algorithm. A text file, 'Median.txt', including integer numbers ranging from one to 10000, which are not sorted is provided. You should behave with this files numbers as a stream of numbers that arrive one-by-one. Assume that x_i is the i^{th} number of the file and m_k is the median of $x_1, x_2, \dots, x_k$. (So, if the k is an odd number, then m_k, is the $\frac{k+1}{2}$ number in the sequence. And if k is an even number, then the median is the $\frac{k}{2}$ number of the sequence.)

We have to find the summation of these 10000 medians and print only the last 4 digits of it.in other words, you have to find the answer of the following summation:

$$\sum_{i=1}^{10000} m_i \ mod\ 10000$$

And at the end, you should compare the algorithms that are based on heaps and trees.

Heap-based algorithm

First, we explain the heap-based algorithm. for the beginning, we start by explaining what is heap first, then we say the heap-based algorithm, after that we explain the heap-based algorithm and the way that we used python implementation of heap and how did we change it for our use.

What is Heap

Heap[1] is a very useful data-structure that is used for retrieving the max/min element (based on using which kind of heap) at O(1). The interesting part of heap, is about insertion and popping elements. It inserts and pop elements in O(log n). And its creation from an array is accomplished in O(n). but how?

It stores data in a list[2] but actually uses the binary-tree structure for storing them. Each node that is placed in the i^{th}index of the array, its children are stored in 2i + 1and $2i + 2$ indices. And Each node is less than its two children.

[1] Heap or priority queue
[2] List or Array

By this approach, the first node of the tree, the root, is the node with the least value. So, every time that anyone wants to get the minimum number, this data structure returns the root. So, this is how the retrieving operation is done **in O(1)** time.

But what about the insertion? For the insertion, it inserts the input element in the last place of the array which is not filled yet (or insert to the end of the list). Then we apply '**bubble-up'** operation. What is bubble-up? We start by the last node of the array, and compare it with its parent; if its value is more then it's parent value, we replace them and recursively call this function for this parent node again until we reach to the root element, that's where we stop. Otherwise, we stop the operation and return. By this approach, the number of recursive calls that we have, in the worst case, is as much as the depth of the tree. And because we store elements in a binary tree which is **always balanced** because whenever we insert a node to the array, we insert it to the first index that is empty. So, we don't leave any index empty and move to the next. So, if we don't fill all nodes of a layer, we won't move to the next layer. So, the insertion is done at **O(log n)** time.

About delete (pop) operation, for this operation, we simply delete the element at the beginning of the list, the root of the tree, and put the last element of the tree in its place. Then, we apply an action which is called 'bubble-down'. For this, we compare the node (here the node is the root of the tree at the beginning) with both its children, if its higher that its children, we replace that element with the least element among its children and call this function recursively; otherwise, we stop the operation and return. So, in the wors possible scenario, we have recursive calls as much as the depth of the tree, which as we explained before, the tree is always balanced, so its depth is always $\left\lfloor \frac{n}{2} \right\rfloor$. So the delete operation, is done in **O(log n)** time.

At last, we talk about its creation (heapify) time for n elements. We don't need that here, but we briefly mention it. It is done in O(n) time. It stores elements in the array, then it applies the 'bubble-up' operation for elements in the second half of the array, starting by the last element in the array and finishing with the element that is in the $\left\lfloor \frac{n}{2} \right\rfloor$ index. By writing we can achieve this number, but this is not in our context, so we don't bring it. But the operation is done in **O(n)** time.

Algorithm Description

For median our goal, which is maintenance, we use two heaps, one max-heap and one min-heap which the max heap is used for storing elements of the lower half of the algorithm and min-heap is used for storing upper half elements. For this purpose, we use "*heapq*" python package. This package integrates with the python list data structure and stores numbers (elements) in it and have methos for applying insert, pop, generate (heapify) and …. Always the element at the zero index of the element, is the root of the tree. But there is only one problem with it which we solve it with a little trick. The problem is that this package can only generates and maintains min-heap. How to sole this problem? We use the following trick to overcome this problem. For max-heap, we use the same approach, but we use the negative numbers for it. For example, if our input is 26, the thing that we insert to the heap, is -26. This won't make us any problem because we don't have negative numbers that make us confuse whether the number was originally negative or not. So, by this

approach, we convert the min-heap to the max-heap. So, now we are good to go with the solution that we provided in the beginning of this section.

So, in our solution, we have to provide an approach for retrieving the median element and inserting element. Our approaches are as follows:

- **Insertion:**
  For insertion, we compare the length of these two heaps that we have. If there are equal or the lower-half (lower half heap) has less length, we add the number to the lower half heap, which is a max-heap, by using the "*heapq.heappush*" function. Then we balance the halves (heaps) if needed. (In code, this is done by calling "*__balance_if_needed*" function.) what does this function do?
  This function checks the maximum number that we have in the lower half (first element of the max-heap that we have) is less than the leas number that we have in the higher half (first element of the min-heap), we pop items from their heaps using the "*heapq.heappop*" function and then insert them to the other half (heap) by using "*heapq.heappush*" function. So, by starting this function from the begging and having the condition for adding the elements at the beginning, we always maintain the correct elements in lower half and higher half. and we always know that the length of these halves, will never have absolute difference higher than one. Because the moment that one of theme's length gets one more that the other, at the beginning of the insertion process, the new number will be added to the heap with lower length, so their length will be the same again. And all the times, the lower half, has equal or more elements in it. And the root of the first half heap, lower half, which first element of the lower list, is the median.
  What about the condition that we put in the code at the beginning of the "*__balance_if_needed*" function? What is that for? That because the only way that we can have zero length for any list, is at the time that we just have inserted the first element to the list.
  How to proof that each half, contains only the lower half and the new number will not ruin that? Easy. We proof it by induction to the input number. We know that for the first input, it will be inserted to the lower half and it is ok. The second element will be inserted to the higher half at first but at the "*__balance_if_needed*" function, the element at the higher half and the lower half, will be changed if needed. So, this was our induction step. Now lest see if we have some elements at each half (with length of $\left\lfloor \frac{k}{2} \right\rfloor$ for higher half and $\left\lceil \frac{k}{2} \right\rceil$ for lower half) the next element will not ruin the halves correctness. We have two scenarios on the length of the halves: he length of halves are equal or their length are not equal. For the first scenario, their length is equal, the new element will be inserted to the lower half, then the highest element of the lower half will be updated. Then we will compare the lower half maximum element and the higher half minimum element and change their place if needed. And hence we know that they are already correct, the only case that can make us any problem is when the new number should be in the higher half. in that case we change the maximum number of the lower half and the minimum number of the higher half and it will

be converted to the correct form because the number that we remove from the higher half, is the least number due to sake of heap.
With other scenario, that if the length of these two halves are not equal, as we know from before, we know that the length of the lower half is more because it has more priority at the insertion time, so the new element will be inserted to the higher half. then the only case that can makes us any trouble, is the case that this element actually belongs to the lower half. in that case, in the "*__balance_if_needed*" function we change it with the maximum number of the lower half. and because we are using heap data structure, the first number is what we want. Then in this case we won't have any problem too. So, we won't have any problem at all and due to sake of the induction our proof is completed for the k element if the previous are right. Then the proof is complete.

- **Retrieving median:**
  For retrieving median, we only need to get the first element of the lower half. because the lower half contains the middle number all the time. If the summation of these two lists length are equal, then the lower half has the maximum number among the lower part, so by the definition of this question, it contains the median. On the other case, if the summation of these two lists is odd, then the first array length because always is greater than or equal the other list, will contain the median. So, all the time, we can retrieve the first element of the lower half which due to the storing way of the *heapq* package and our format, is the first element. (but as was explained before, we keep the negative of numbers in the lower half)

Tree-based Algorithm

In this algorithm we use the **BST**[1] data structure and always keep our tree balanced somehow. We could use any other data structures that keeps the BST balanced, but we rather to implement our models to be utilized for retrieving the median each time easily.

First, we briefly introduce the BST, which we know from the course, then we will move forward to the algorithm description.

BST

STD is a tree data structure that each node can has at most two children and can only have one parent node. BST has some interesting features that were introduced during class and books like their average height is O(log n) and since their average height is that, operations like insertion, deletion and search can be done in O(log n). Because this data structure was introduced, we omit further descriptions and go to the algorithm description.

[1] BST stands for Binary-Search-Tree

BST-based Algorithm

In this example we use the BST data structure. So, first we implement the node calls that stores a value, parent node and its children. There is nothing special for this class, let's move to the BST class implementation.

We have a node called root. That's how we store the tree. We also keep count of nodes at each side of the root. Then in the insertion time, we generate the root if we don't have it at the moment. Otherwise, we add the number of left side nodes or nodes count on the right side depends on the value that we want to add and the value that the root has at the moment. Then we add the element to the tree just like the BST. We search for the number, then when we get to the node that we have to move to its right or left child and it doesn't have it, we add that node with the value that we want. Then same as the previous approach, we call the "*__rebalance*" function. This function checks whether the left-side counts are more than the nodes counts on the right side or not. If there are more nodes on the left side, then then we call "*__rotate_right*" function and decrease left side nodes count and increase right side nodes. But if the number of left side nodes count plus one is less than the number of nodes on the right side, then we call "*__rotate_left*" function and decrease right node counts and increase left node counts. And by a similar proof, the difference between the left side node counts and the node counts on the right side will be same or node count on the right side is only one more than the left side node count. The rotate functions are just similar to the rebalance function of the tree. For example, for the "*__rotate_right*" function, we find the maximum element from the left side of the root (the maximum number between all numbers that are less than root) and then set it as the root. For the "*__rotat_left*" function, we do the exact same thing but we find the minimum number (element) that is higher than the root. We handled every possible state and handled them carefully like having the successor as the root's child or having no children and ….

Now we move to the most important part how to get the median at each step? By this structure that we have here, always our root is the median of the numbers that have been inserted. So at each time we only need to access the root and its element. So this action will be performed in O(1). But what about the insertion time? Is it still feasible at O(1)? No, unfortunately this is the problem that this approach has. The problem is that the insertion completely depends on the BST that will be structed, which is massively depends on the inputs and their orders. On average, the height of the tree is in O(log n) but in this example we don't know it for sure and it completely depends on the input stream. But if we consider the height of the tree as h, then the insertion order at this time will be at O(h) in the worst case. So, if we consider the worst case every time and we consider the height of the tree from O(log n), the cost of insertion O(log n) on average and O(n) is in the worst case.

Comparison

Now, we have to compare these two algorithms and see the results. First, we read the file and store numbers in a list called "*numbers*". This action is done because we don't want to consider IO bounds for our implementation and we only want to compare these two algorithms and we don't want to consider other things as much as we can. So, we try to reduce other things effect as much as we can. Then we run a simple test for the Heap-based algorithm approach and check whether it works correctly or not. Of course, we had done many previous checks before that, but just for illustration, we added this to the file.

Then we apply the file's input for the Heap-based algorithm and print the result. (Due to the previous checks that we have had, we know that the answer is correct, we even get this answer in the BST-based algorithm too) but in addition to the result, we store the execution time in a variable called "*heap_exec_time*".

Now we go to the next part, in which we do the exact same thing with the BST-based algorithm. first, we apply a simple test on it (as same as the Heap-based algorithm, we had done many other tests before that too, be this is the sample that we have put there for the illustration). Then here, we also draw the graph of the BST tree that is constructed with the tests and then check whether it works correct or not. And this graph shows that our algorithm works correct. Then we move to the next part and apply the algorithm for the file input and we get the result and the exection time that we store in a variable called "*bst_exec_time*". Then when we print the result,we see that the result is same.

In the last part, we compare the execution time and answers of these two approaches. As we have noticed, the answers are same and the "have same result: True" is printed successfully. So, the first phase (which was about the correctness of the answers) was successful. At the end, we move to the time comparison. In time comparison, the Heap-based algorithm works better very much. The reason is obvious somehow, even if the insertion of both algorithm is done in O(log n) which about the Heap-based algorithm we know this for sure, but for the BST-based algorithm the average case (somehow even the best case) is O(log n). So, we might see that the Heap-based algorithm works better. And when we see the codes result for comparison of these two algorithms time, we see that the **Heap-based algorithm has been better** quit much.

Huffman Algorithm

This part is for the **Huffman algorithm**, which is a **greedy** algorithm. First, we go for the question and explain it and then we explain the algorithm and our implementation and after that we go through the results and explain them. Somehow in the same order as the previous part.

Question

In this question, we were asked to implement the Huffman algorithm which is a greedy algorithm. there is file, which is provided for which contains number of the symbols in the first line and in all other lines, there is the weight of symbols consecutively. (for example, after the total number of symbols and in the next line, the weight of the first symbol is provided, in the next line, the weight of the second symbol is provided and …) Our goal in this part is to apply Huffman algorithm for this file inputs. The requirements are as follows:

- What is the most length that a symbol's code has in Huffman algorithm?
- What is the least length that a symbol's code has in Huffman algorithm?

We have to implement the Huffman algorithm and answer these questions.

Algorithm

This algorithm is considered as a **Greedy** algorithm. at the end of the algorithm description part, we will explain why. For this part, first, we start by describing Huffman algorithm and then we explain its implementation.

Algorithm Description

The algorithm description is as follows. In this algorithm, we make a tree and each node is the leaf of this tree and on each edge of this tree, there is a bit (zero or one). Then when we want to specifies each symbol's encoded binary string[1] and then we find the path from root to the leaf node that contains that symbol. Then the bits on that path, is the final encoding that we have for that symbol. But how to generate this tree? First, we compute the weight of each symbol (character) and make a node for each of them and use symbol inside each node and set the node's weight[2] to the symbol's weight. Then we make a heap (min-heap) using these nodes. Then in a loop, at each iteration, we pop first two nodes (nodes that have the least wight among other nodes) and then make a super node that is the parent of these two nodes (the node with higher weight is the right child of this super node and the other node, node with the less weight, is the left child of this super-node, or we can simply apply the reverse approach and change the children's place, it won't make any

[1] Binary string is a string containing only zero's and one's
[2] Weight and frequency are used interchangeably in this part

differences) then at the end insert that super node to the heap and set its weight to the summation of its two children's weight. And we repeat this operation until there is only one element left in the heap. Then that node (super node) will become the root of our tree. Then we can have this simple rule, we put zero on each left child's edge (the edge that connect a super node to its left child, has zero on it as the edge bit value) and set the right child's edge to one (set the bit one to the edge that connects each super node to its right child) or we can change the rule for zero and one (zero for the edge that connects a super node to its right child and one for the edge that connects a super node to its left child), it won't change any difference.

Now we can simply see why this algorithm is considered as a **Greedy** algorithm. because as wee saw, at each iteration we get the elements with the least amount of weight. So, at each iteration we are checking for an element with the least parameter (weight parameter here), and that is the reason that makes this algorithm a **Greedy** algorithm.

With this implementation, the time complexity of this algorithm is of **O(n log n)**, because it takes O(n) time to make the list of nodes and apply the **heapify** function on it. Then it has a loop over the heap, and at each iteration, it removes two element from the heap and add one element, so we have n-1 total iteration at this loop, which at each iteration we pop two element, which is done in O(log n) and push one element, which is also done in O(log n). So, the total execution time of this algorithm is in **O(n log n)**.

Implementation

The implementation consists of two important parts, the '*Node*' class implementation and the '*HuffmanAlgorithm*' class. So, we are going to explain each part consecutively.

- **Node class**:
  In this class, we implemented the Node class which is the node and super node part. This class consists of some properties. These properties are '*val*', which is the symbol that each node has, '*freq*' which is each nodes weight (frequency), '*parent*' which contains each node's parent, '*left*' and '*right*' which are used for storing each node's left and right child consecutively and a '*code*' property which contains the final code of each node based on their place in the final tree. We could ignore the last property, but because we didn't want to generate each node's code at every time that we wanted them, we compute it once and use it as many times as we want.
  Then, this class contains setter and getter function for all these properties, which have nothing to be explained, they simply just set and get the value of the described properties.
  But there are some other functions, that we explain them in following part. These functions are as follows:
    - **'get_graph_repr' function:**
      This function is used to get the string that is going to represent this node in the graph hat we are going to plot for the tree that is generated by this algorithm.

- **'has_value' function:**
  This function is for checking that this node is whether a super node or not. Returns True if this node is a leaf (contains symbol) and False if this node is a super node and doesn't have a symbol.

- **'*__str__*' function:**
  This is the function that is used whenever we want to convert this node to a string ('*str'* type in python), which uses the '***get_graph_repr***' function.

- **'*__repr__*' function:**
  This function is used whenever we put a variable of this type in a '*jupyter-notebook*' and it represents its value for us.

- **'*__lt__*' function:**
  It is used for comparing two nodes with each other. It is used by the heap whenever it wants to insert and pop any elements.

That is all with the node class. Not very complicated and doesn't contain any special logic of the algorithm, it is used as a way to store data.

- **HuffmanAlgorithm class:**
  This class contains the algorithm logics and implementations and some other functions for getting the max length and the min length of a code among symbols. All these functions are static functions. Now, we briefly explain all these functions and what they do. These functions are as follows:
  - **'*generate_tree*' function:**
    This function is used for generating the tree of this algorithm. This function, gets a list containing two elements at each index, which the first element is the symbol value, which must be converted to string or a type that can be converted to string and the second element is each symbols weight in the file. It generates a list of nodes for these values and their weights and then by using '*heapq.heapify*' function (which as we explained before, this library uses min-heap), these nodes list, will be converted to a heap data structure. (this function applies changes **in-place**) then as the algorithm description explained, at each iteration, until the heap contains only one element, it pops from the heap twice and because we use min-heap, the elements that we get are the elements with the least weight (frequency) among other elements (nodes). Then by the explained rule, we combine them by generating a super node and insert this super node to the heap using the '*heapq.heappush*' function. And at the end, we call the '**generate_tree**' function which will set each nodes code in the constructed tree by getting only the root of that tree. We will explain this function in the next part.

- **'*set_tree_codes*' function:**
  This function is used for setting the code of each node in the constructed tree. This function gets the root of the tree as an input. This function was meant to be implemented recursively, but for that we needed to have the string that represents the path that we have traveled from the root to that node, and we didn't want to put that in the function parameter because this function can be seen from outside of this class, so we implemented another function and the logic is there and this function names is '***__set_node_code***'. We only call that function here.

- **'__set_node_code' function:**
  This function is used for generating the code of each node, and it recursively calls itself but for children of the input node and always update the path when it wants to call itself for the required node children.

- **'*generate_nx_graph*' function:**
  This function is used for making a **'*networkx graph*'** of the tree and plot it after the tree gets constructed and node's code is constructed in order to check the algorithms behavior easier. So, this function just converts our tree structure by only getting the root of the tree to network graph.

- **'*plot_tree*' function:**
  This function is used for drawing the generated graph in the previous part and displaying it.

- **'*__depth*' function:**
  One thing that the question requires us to do, is to find the symbol with the least and the longest and shortest code length in the generated codes. And for this, as we know per algorithm description, each leaf node's length, depends on its depth in the generated tree. But if we wanted to implement these functions separately, we would have two functions that were different in one line at most and were totally duplicated, so we implemented the logic here, and by getting a function (a comparator function) as an input, which this function, can be python's default '***min***' function or '***max***' function and by that this function returns the maximum or minimum code length in the constructed tree. This function behaves recursively. If the node is a leaf, it returns that node code length; otherwise, it call's itself for each of its children and then store the results in a list in and then, at the end, it compares these results by the function that it gets as the input and return the result. This function is private in order to be hidden from other classes to access it. And we only defined two function that can be accessed from outside of this class and we will explain them next.

- **'*get_min_code_len*' function:**
  This function uses the '***__depth***' function with *python*'s '***min***' function as input in order to get the **minimum code's length** in the constructed tree.

- **'*get_max_code_len*' function:**
  This function uses the '***__depth***' function with *python*'s '***max***' function as input in order to get the **maximum code's length** in the constructed tree.

As we have explained, this class contains the logic of the algorithm and by calling its static functions, it will apply the **Huffman** algorithm for the given input.

Results

In this part, we use the algorithm implementation that we have explained in the previous part and use it to solve the question for the provided input. Codes are located in the '***sample.ipynb***' file. First, we import the '***HuffmanAlgorithm***' class that we have implemented. Then at the first step, we generate some numbers and check whether our algorithm works correct or not. (many others test and debugs were applied and the code was debugged, but here we just put this simple sample to just say that we have tested our algorithm and codes) As you can see, the algorithm for the generated input works fine and the tree plot is represented for the template.

Now, we move to the main part, which we apply the algorithm for the provided inputs in the file. First, we read the file and store its values in a list. We store the symbol as string and the symbols weight as integer in our list in order to make us simply able to construct the tree and codes for the provided input. Then after reading from file and storing them in the list in the required format, we run the algorithm for this input and measure the execution time. At the end, after the tree construction, we print the execution time.

In the next part, as the question requires, we print the minimum and maximum length of a character in the generated codes and we also print the difference of the length of the minimum and the maximum in addition.

At the end, we make the graph of the constructed tree and save it in a file with the '*huffman-code-tree.png*' name. we didn't plot that graph image in the notebook file, because it was too heavy and big, it wasn't clear at all and it was very time consuming. But the image has been provided in the uploaded files.

References

1- Classes and TA classes
2- **CLRS** book
3- **Michael T. Goodrich, Roberto Tamassia Algorithm Design and Applications** book
4- **Network** website
5- **Geeks-for-Geeks** website
6- Many other references that are unfortunately not available or remembered now that were used for the coding parts