Cairo University
Faculty of Engineering
Credit Hour System

SBEN429 Biomedical Data Analytics
Assignment 6

Name: **Sandra Adel Aziz Gebraiel**

ID: **1180059**

Date: **11/01/2022**

Build Heap (Array to Heap):

1) Explanation of Code:

In order to turn an array into a heap, provided that we have the parent-children connections through indices, we have to target all edges that may violate the heap property (top node greater than bottom node for min heap) and swap their nodes, and tracking each swapped node recursively until we are sure that the heap property is satisfied in all the nodes of this sub-tree.

As all the leaf nodes have no children, then they have no edges which may violate the heap property, so we begin checking from the level before the last (at depth 1) till the root at the first level, targeting greater and greater sub-trees, solving any violation of the heap property (by sifting down to the leaves of this sub-tree) until we reach the result that whole tree is valid as heap.

Since each level i has a number of nodes equal to n/(2^i), then the expression floor(n/2) would approximately give the rightmost node in level of depth 1 to start handling. Sometimes, it gives the leftmost node in the last level, which already satisfies heap property. Therefore, we loop on nodes from indices floor(n/2) to 0 in a zero-indexed array.

Each violating edge is solved by calling sift_down function, which gets the left and right children of this nodes, and if any of them exists and has a value less than this parent, we swap their values, save their indices in the list of swaps and recursively track down the swapped nodes down the tree, in the direction of the smaller child (min heap), until the whole sub-tree satisfies the heap property.

In an array-saved-heap, we can find the parent-children connections on the fly through arithmetic calculations of indices. For a zero-indexed array, for a node i: 1) index of left child is (2xi) + 1 and 2) index of right child is (2xi) + 2

COMPLEXITY: O(n) Actually
We call sift_down, of complexity O(logn) n times, so the complexity should be O(nlogn). This is true, we calculate the complexity of sift_down based on the worst-case-scenario in the context of an individual case, not in context of a totality of n operations and in build_heap, we call this procedure n times.

To illustrate, most of the called nodes are close to the leaves, so they have complexity of value much less then O(logn), and only the root node is the one of complexity O(logn). According to Banker's method, the cost of n sift_down operations can be rooted down to 2n (summing the number of nodes at each level x cost of sifting down nodes in this level).

So, amortized cost of this procedure is O(1), called in times in build_heap --> O(n)

2) Pseudocode:

LeftChild(i):

Return (2xi) + 1

RightChild(i):

Return (2xi) + 2

SiftDown(i, Array, Swaps):

MinIndex ← i
Size ← Length of Array

L ← LeftChild(i)
If L is in the heap and Array[L] < Array[MinIndex]
    MinIndex ← L

R ← RightChild(i)
If R is in the heap and Array[R] < Array[MinIndex]
    MinIndex ← R

If i is not equal to MinIndex:
    Add (i , MinIndex) to list of Swaps
    Swap Array[i] and Array[MinIndex]
    Swaps ← SiftDown(MinIndex, Array, Swaps)

Return Swaps

BuildHeap(Array):

Initialize a Swaps as an empty list

For i from floor(n/2) down to 1:
    Swaps ← SiftDown(i, Array, Swaps)
Return Swaps

Heap Sort Algorithm:

1) Explanation of Code:

Heap Sort is a fast-sorting algorithm (of complexity O(nlogn)), which is stable (worst and average cases are O(nlogn) in comparison to Quick Sort whose average case is O(nlogn) while its worst is O(n^2)) and is space-efficient as it sorts in place using no additional memory, using the array implementation of a heap.

Firstly, the array is turned into a heap (all edges of tree satisfying heap property) as explained in Build Heap section, but in this case, this is a max heap, so each parent must be at least the value of its children.

Secondly, we loop n-1 times on the array. In each time, we swap the value of the root, which is maximum, with the value of the last node (rightmost leaf), putting it at the end of the array as we are sorting ascendingly. We decrease the size of the heap by 1, ignoring the last element which is already in its right place in the sorted array, and maintaining the heap property in the rest of the array, a fact that is violated if we take the last element into account.

Lastly, we heapify (sift down) the new root so that the heap property is satisfied in the new tree. Here, sifting down occurs in the direction of the largest child as this is a max heap, and the heapify function takes size as a parameter, as it changes in each iteration and is not just the size of a

We loop n-1 times, so we do not place the minimum value ending up at the root in the (n-1)th iteration at the end of the array. As a result, we have an ascendingly sorted array with lesser complexity than selection sort. This optimization is illustrated as in both, we iterate n times over the array elements, but the second iteration of finding the maximum value in selection sort is avoided by using a smart data structure such as max heap where the maximum value is always at the root, the first element in the array.

COMPLEXITY: O(nlogn)
Since we know that the amortized analysis of build_heap is of cost O(n) from explanation in Build Heap section, we still loop at most n times over the array, calling heapify (O(logn)) n times. And the complexity is already asymptotically optimal as this is a comparison-based sorting algorithm

2) Pseudocode:

LeftChild(i):

Return (2xi) + 1

RightChild(i):

Return (2xi) + 2

Heapify(i, Array, Size):

MaxIndex ← i

L ← LeftChild(i)
If L is in the heap and Array[L] > Array[MaxIndex]
    MaxIndex ← L

R ← RightChild(i)
If R is in the heap and Array[R] > Array[MaxIndex]
    MaxIndex ← R

If i is not equal to MaxIndex:
    Swap Array[i] and Array[MaxIndex]
    Heapify(MaxIndex, Array, Size)

BuildHeap(Array):

For i from floor(n/2) down to 1:
    Heapify(i, Array, Length of Array)

HeapSort(Array):

BuildHeap(Array)
Size ← Length of Array

For i from 1 to size - 1:
    Swap Array[1] and Array[Size]
    Decrement Size of Heap
    Heapify(1, Array, Size)

3) Naïve Algorithm and Stress Testing Code:

```
# Naive Algorithm: O(n^2)
def selection_sort(data):

    for i in range(len(data)):
            for j in range(i + 1, len(data)):
                if data[i] > data[j]:
                    data[i], data[j] = data[j], data[i]


if __name__ == '__main__':

    '''
    #input = sys.stdin.read()
    #n, *a = list(map(int, input.split()))
    n = 10**5
    a = [10**9] * n
    heap_sort(a)
    for x in a:
        print(x, end=' ')
    '''

    while 1:
            a = []
            n = r.randint(1, 10**3)
            print(n)
            for i in range(n):
                a.append(r.randint(1, 10**9))
            print(a)

            res1 = selection_sort(a)
            res2 = heap_sort(a)

            if res1 != res2:
                print('wrong answer'+ ' ' + str(res1) + ' ' + str(res2))
                break
            else:
                print('OK')
```

Resources:

MOOC: Data Structures Slides - Week 3