Homework Batch II: Trees and Algorithms

Davide Basso - SM3500450

Exercise 1

Let H be a Min-Heap containing n integer keys and let k be an integer value. Solve the following exercises by using the procedures seen during the course lessons:

(a) Write the pseudo-code of an in-place procedure RetrieveMax(H) to efficiently return the maximum value in H without deleting it and evaluate its complexity.

```
def RetrieveMax(H):
    mid <- floor(H.size/2)
    max <- mid
    for i <- mid + 1 to H.size:
        if max < H[i]:
            max <- i
        endif
    endfor

    return max
enddef
```

In this algorithm we are dealing with a Min-Heap represented thorugh an array. We know from theory that in this way root node will be placed in 1^{st} position and each left and right child of $i-th$ node will be then respectively found $2i-th$ and $2i+1-th$ position. Basically what we are doing is to exclude *a priori* all the nodes that have children (and so that are $\leq$ than other nodes due to the min-heap property) and then retrieve the maximum through a linear scan. The function returns the index of the maximum value.

In order to evaluate the complexity of this algorithm we just have to count the number of iterations performed by the for loop (all the operations inside it cost $\theta(1)$):

$$n - \lceil \frac{n}{2} \rceil \approx \frac{n}{2}$$

So we have that the algorithm belongs to $\theta(n)$.

(b) Write the pseudo-code of an in-place procedure DeleteMax(H) to efficiently delete the maximum value from H and evaluate its complexity.

```
def DeleteMax(H):
    max <- RetrieveMax(H)
    swap(H, max, H.size)
    H.size <- H.size-1
    i <- max

    while not (is_root(i) or H[parent(i)] <= H[i]):
        swap(H, H[i], H[parent(i)])
        i <- parent(i)
    endwhile
enddef
```

Maximum value will be for sure a leaf node, so in order to remove it we can swap it with the last element of the heap and decrease by 1 the heap size itself. However we need to be sure that the binary heap property is still satisfied. To do so we check if the parent is bigger than the newly changed child, if so we swap them and push the problem upwards until root is reached.

We can finally compute the overall complexity, which for sure will be strictly connected to *RetrieveMax(H)* one, and in fact it's equal to:

$$\theta(n) + O(log(n))$$

where the second term is due to the bin-heap property checking (at most done $height(H) = log(n)$ times).

(c) Provide a working example for the worst case scenario of the procedure DeleteMax(H) on a heap H consisting in 8 nodes and simulate the execution of the function itself.

Let's take `H = [1,2,30,4,5,60,70,8]`. By applying the previously defined algorithm we find out that the maximum is placed in position 7. Then the swap with the last element of the heap is operated. Now we have that `H = [1,2,30,4,5,60,8]` but bin-Heap property is no longer satisfied so the algorithm procedes with the swap between parent (30) and child node (8). Doing so the heap property is fixed and the final result is: `H = [1,2,8,4,5,60,30]`.

Exercise 2

Let A be an array of n integer values (i.e., the values belong to $\mathbb{Z}$). Consider the problem of computing a vector B such that, for all $i \in [1; n], B[i]$ stores the number of elements smaller than $A[i]$ in $A[i + 1; \dots ; n]$. More formally:

$$B[i] = |\{z \in [i + 1; n] : A[z] < A[i]\}|$$

(a) Evaluate the array B corresponding to `A=[2,-7,8,3,-5,-5,9,1,12,4]`.

The result is `B = [4,0,5,3,0,0,2,0,1,0]`.

(b) Write the pseudo-code of an algorithm belonging to $O(n^2)$ to solve the problem. Prove the asympotic complexity of the proposed solution and its correctness.

```
def smallerThan(A):
    B <- array[A.size, default = 0]
    counter <- 0
    for i <- 1 to A.size:
        for j <- i+1 to A.size:
            if A[j] < A[i]:
                counter <- counter + 1
            endif
        endfor
        B[i] <- counter
        counter <- 0
    endfor

    return B
enddef
```

Since requirements for this exercise state that the algorithm needs to $\in O(n^2)$ we can basically just operate, for each index of the array A, a linear scan of the elements placed after the index itself in order to check if any of those is smaller than the current element at position i.

The complexity, following this procedure and considering the fact that all assignment and increment operations cost $\theta(1)$, will be:

$$\sum_{i=1}^{n}\sum_{j=i+1}^{n} 1 = \frac{n(n-1)}{2} \implies \in \theta(n^2)$$

And this result actually satisfies the required complexity.

Correctness of the algorithm is pretty self explanatory and proved just by following all the described steps.

(c) Assuming that there is only a constant number of values in A different from 0, write an efficient algorithm to solve the problem, evaluate its complexity and correctness.

Let k be the total number of values in A that are different from 0. The strategy I opted to follow in order to enhance the complexity of the previously presented algorithm was to copy all the k elements different from 0 into an auxiliary array AUX and as first step to apply previous algorithm on this newly generated array. Then I operated another linear scan of the input array using a for loop and, taking advantage of some auxiliary indexes, I computed resulting values of the output array B following the procedure described in the pseudocode below:

```
def improved(A, k):
    AUX <- array(k, default = 0)
    i_AUX <- 1
    zeros <- A.size - k
    neg <- 0
    for i <- 1 to A.size:
        if A[i] != 0:
            AUX[i_AUX] = A[i]
            i_AUX <- i_AUX + 1
        endif
        if A[i] < 0:
            neg <- neg + 1
        endif
    endfor

    B_AUX = smallerThan(AUX)
    i_AUX <- 1
    B <- array(A.size, default = 0)

    for i <- 1 to A.size:
        if A[i] > 0:
            B[i] = zeros + B_AUX[i_AUX]
            i_AUX <- i_AUX + 1
        endif
        if A[i] = 0:
            B[i] = neg
            zeros <- zeros - 1
        endif
        else:
            B[i] = B_AUX[i_AUX]
            i_AUX <- i_AUX + 1
            neg <- neg - 1
        endelse

    return B
enddef
```

It's possible to prove the correctness of the algorithm by taking into account a brief example. Let's consider the array `A=[2,-1,3,0,-1,-2,0,7]`, so $k = 6$ and `zeros=2`; from first loop we have that `AUX=[2,-1,3,-1,-2,7]` and `neg=3`; after this step we can compute B_AUX which is equal to `B_AUX=[3,1,2,1,0,0]`. Then we can procede with the second loop:

- since `A[1] > 0`, `B[1] = zeros + B_AUX[1] = 2 + 3 = 5`; still we have to increase `i_AUX to 2`
- `A[2] < 0`, `B[2] = B_AUX[2] = 1`; let's decrease `neg to 2` and increase `i_AUX to 3`
- `A[3] > 0`, `B[3] = zeros + B_AUX[3] = 2 + 2 = 4`; still we have to increase `i_AUX to 4`
- `A[4] = 0`, `B[4] = neg = 2`; we decrease `zeros to 1`
- `A[5] < 0`, `B[5] = B_AUX[4] = 1`; we decrease `neg to 1` and increase

`i_AUX to 5`
- `A[6] < 0`, `B[6] = B_AUX[5] = 0`; we decrease `neg to 0` and increase `i_AUX to 6`
- `A[7] = 0`, `B[7] = neg = 0`; we decrease `zeros to 0`
- `A[8] > 0`, `B[8] = zeros + B_AUX[6] = 0 + 0 = 0` and we increase `i_AUX to 7`

After all these steps, we have that `B=[5,1,4,2,1,0,0,0]`, which is the correct result.

Complexity in this case is equal to $2*\theta(n)+O(k^2)$ (respectively related to the two loops and *smallerThan* function execution). Since k is a constant number and independent from n, $O(k^2)$ becomes negligible and so the resulting complexity is $\theta(n)$.

Exercise 3

Let T be a Red-Black Tree.

(a) Give the definition of Red-Black Trees.

Red Black Trees are Binary Search Trees satisfying the following conditions:

- Each node is either a red or a black node
- The tree's root is black
- All the leaves are black nil nodes
- All the red nodes must have black children for each node x
- All the branches from a node x contain the same number of black nodes

(b) Write the pseudo-code of an efficient procedure to compute the height of T. Prove its correctness and evaluate its asymptotic complexity.

In order to find the height of an RBT what I thought to do was to traverse it, starting from root node, and to compute recursively the height of the right and left subtree by taking the maximum of these two and adding 1 to it. In this way when leaves are reached the algorithm returns 0 but rolling back the recursion calls let's us retrieve the actual height of the tree.

This procedure is described below:

```
def height(node):
    if node = NULL:
        return 0
    endif
    left_H <- height(LEFT_CHILD(node))
    right_H <- height(RIGHT_CHILD(node))

    return max(left_H, right_H) + 1
enddef
```

Correctness and complexity of the algorithm can be proved by taking a look at the pseudocode. In fact as explained before, this approach consists in computing the height of each node in the RBT and so the expected time complexity will be:

$$T(n) = 2 * T(n/2) + \theta(1)$$

Since we are dealing with an RBT we can assume that the tree is almost balanced and so we can retrieve the precise final complexity:

$$T(n) \leq \sum_{i=0}^{h-1} 2^i = 2^h - 1 \approx n \implies \in O(n)$$

(c) Write the pseudo-code of an effient procedure to compute the black-height of T. Prove its correctness and evaluate its asymptotic complexity.

Differently from previous case, we can apply a smarter approach in order to compute the black-height of an RBT. In fact, due to the last property specified in point (a) of this exercise, we know that in each branch of the tree the number of black nodes is the same. So to reduce time complexity we can consider to traverse only one among all the branches (in this case only left-most one).

```
def black_height(node, counter):
    left_child <- LEFT_CHILD(node)

    if color(left_child) = black:
        counter <- counter + 1
    endif

    if left_child = NULL:
        return counter
    endif

    black_height(left_child, counter)
enddef
```

Complexity in this case will be proportional to the height of the tree, since at each recursive step we are traversing in depth the tree itself and from theory we know that $h \leq 2log(n+1), n = \{\#ofnodes\}$ we can state that time complexity $\in O(log(n))$.

Exercise 4

Let $(a_1, b_1), \ldots, (a_n, b_n)$ be n pairs of integer values. They are lexicographically sorted if, for all $i \in [1; n-1]$, the following conditions hold:

- $a_i \leq a_{i+1}$
- $a_i = a_{i+1}$ implies that $b_i \leq b_{i+1}$.

Consider the problem of lexicographically sorting n pairs of integer values.

(a) Suggest the opportune data structure to handle the pairs, write the pseudo-code of an efficient algorithm to solve the sorting problem and compute the complexity of the proposed procedure.

A possible data structure to handle properly the pairs and provide a solution to the sorting problem could be an array of pairs (where pairs could be implemented either as an array of 2 elements, the choice that I've taken, or defining a new complex datatype). To do so an implementation of Lexicographical ordering is needed in order to pass it as total order of a sorting algorithm that work through comparison. Practically, these are the steps to follow:

```
def lexicographical_order(a,b):
    return a[0]< b[0] or (a[0]=b[0] and a[1]<=b[1])
enddef

def soritng_lexicographically(A, lexicographical_order):
    return heapsort(A, total_order = lexicographical_order)
enddef
```

Side note: here we are referring at heapsort implementation done during lectures.

In the solution proposed above I used as sorting algorithm *heapsort* (or equivalently it can be used *quicksort* but there could be some problems concerning complexity in worst case scenarios) which has complexity $\in O(nlog(n))$, and actually is the best reachable time complexity for this kind of algorithms based on comparison (using *insertion sort* for example would have produced same results but with worse complexity, $O(n^2)$).

(b) Assume that there exists a natural value k, constant with respect to n, such that $a_i \in [1, k]$ for all $i \in [1, n]$. Is there an algorithm more efficient than the one proposed as solution of Exercise 4a? If this is the case, describe it and compute its complexity, otherwise, motivate the answer.

In this particular case we know *a priori* the range of values that a_i can assume. We can take advantage from this aspect by using Counting Sort algorithm for sorting first elements of the pairs, moreover doing this in linear time (since complexity of Counting Sort is $O(n + k)$). First of all we need to slightly change the algorithm previously described since we want to separate sorting procedure for a and b elements in the pairs, and this translates in basically define a new total order to sort pairs only w.r.t. a and b. Then we can procede with sorting first with respect to b and then with respect to a.

```
def pair_order_a(a,b):
    return a[0] <= b[0]
enddef

def pair_order_b(a,b):
    return a[1] <= b[1]
```

```
enddef

def new_sorting_lexicographically(A, pair_order_a, pair_order_b):
    heapsort(A, total_order = pair_order_b)
    counting_sort(A, total_order = pair_order_a)
enddef
```

It's clear that this additional information doesn't enhance complexity at all because the algorithm would still be bounded by the complexity of *heapsort.* To take a look at this consideration more in detail:

$$T(n) = O(nlog(n)) + \theta(n+k) \in O(nlog(n))$$

(c) Assume that the condition of Exercise 4b holds and that there exists a natural value h, constant with respect to n, such that $b_i \in [1, h]$ for all $i \in [1, n]$. Is there an algorithm to solve the sorting problem more efficient than the one proposed as solution for Exercise 4a? If this is the case, describe it and compute its complexity, otherwise, motivate the answer.

Contrarly to previous case, Counting Sort can be used to sort both pairs' components. The updated algorithm will be:

```
def new_sorting_lexicographically(A, pair_order_a, pair_order_b):
    counting_sort(A, total_order = pair_order_b)
    counting_sort(A, total_order = pair_order_a)
enddef
```

Finally we can observe that in this case, time complexity is improved (since h and k are constant and independent w.r.t. n):

$$T(n) = \theta(n+h) + \theta(n+k) \in \theta(n)$$

Exercise 5

Consider the *select* algorithm. During the lessons, we explicitly assumed that the input array does not contain duplicate values.

(a) Why is this assumption necessary? How relaxing this condition does affect the algorithm?

The main reason for which this assumption is necessary is because if the input array presents a lot of repeated values then *partition* would occur in its worst case scenario and so overall complexity would be strongly affected by this behaviour (practically would go from linear time to quadratic).

To further explain this aspect, let's take into account an example. If all the elements within the input array are the same, regardless the chosen pivot, while performing *partition* G sub-array (the array in which are contained all the elements that are bigger than the pivot) will be empty and so the next *select* algorithm recursive call would operate on $n-1$ elements and so on. Following this reasoning, the running time for partitioning the array in total will be $\sum_{i=0}^{n}\theta(i) = \theta(\sum_{i=0}^{n} i) = \theta(n^2)$.

(b) Write the pseudo-code of an algorithm that enhance the one seen during the lessons and evaluate its complexity.

In order to solve this problem we can use a *three-way partition* method, which basically instead of subdividing the array into two sub-arrays, computes three parititons; one containing all elements smaller than the pivot, one containing all the ones equal to the pivot and the last one containing all the bigger ones. The pseudocode is as follows:

```
def three_way_partition(A, i, j, pivot):
    l <- i
    r <- i
    eq <- j

    while r <= eq:
        if A[r] > A[pivot]:
            swap(A[r], A[eq])
            eq <- eq - 1
        endif

        else if A[r] < A[pivot]:
            swap(A[l], A[r])
            l <- l + 1
            r <- r + 1
        endif

        else:
            r <- r + 1
        endelse
    endwhile

    return l, r
enddef
```

What still we need to do is to change *select* algorithm in order to deal with the 2 indexes returned by *three_way_partition*.

```
def select(A, l=1, r=|A|, i):
    if r - l <= 10:
        sort(A, l, r)
        return i
    endif

    j <- select_pivot(A, l, r)
    left, right <- three_way_partition(A, l, r, j)

    if left <= i and i <= right:
        return i
    endif

    if i < left:
        return select(A, l, left-1, i)
    endif

    return select(A, right+1, r, i)
```

Let's now think about the actual improvements that this approach comes with. It's interesting to notice that worst case scenario now becomes actually a best case scenario since when all the elements in the array are the same, this method basically performs only a linear scan and immediately returns (notice that both G and S have size 0 in this case) the desired position we're looking for. Moreover, since the main problem is represented by duplicated values that are very likely to be the median of medians, by using this kind of partitioning we are discarding at each recursive iteration of *select* all of those and so they no longer constitute a source of problem for the algorithm.

In this way we ensure that the entire *select* algorithm has a complexity of $O(n)$ even if there is a consistent presence of repeated values.