Homework Batch II

Gonzato Andrea

April 30, 2021

All the pseudo-codes use the one-based numbering for the index of the arrays.

Exercise 1A

The min heap property requires that the parent node be lesser than its child node(s). Due to this, we can conclude that a non-leaf node cannot be the maximum element as its child node has a higher value. So we can narrow down our search space to only leaf nodes.

```
1 DEF RetrieveMax(H)
2     max_element = H[floor(H.size/2)+1]
3     FOR i=floor(H.size/2)+2 TO H.size
4         max_element = max(max_element, H[i])
5     ENDFOR
6     RETURN max_element
7 ENDDEF
8
9 DEF max(a, b)
10     IF (a >= b)
11         RETURN a
12     ELSE
13         RETURN b
14     ENDIF
15 ENDDEF
```

In a min heap having n elements, there are $ceil(n/2)$ leaf nodes. Since line 4 cost 1 and is repeted $ceil(n/2) - 1$ times, the time complexity of RetrieveMax is $\Theta(n/2) = \Theta(n)$.

Exercise 1B

To delete the maximum in the Heap first I need to identify the index in the array of the max value in the heap. Then I delete the max value and replace its value with the last value in the heap. I decrease the heap size and restore the heap property by moving the node up in the tree until its parent is smaller or equal.

```
DEF DeleteMax(H)
    max_element = H[floor(H.size/2)+1]
```

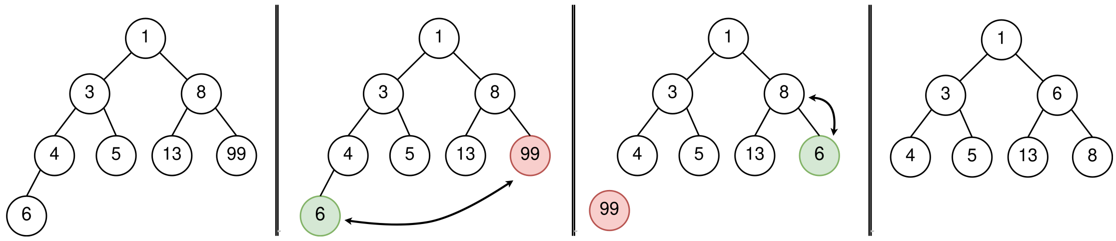
```
3       max_index = floor(H.size/2)+1
4       FOR i=floor(H.size/2)+2 TO H.size
5           max_element = max(max_element, H[i])
6           IF (H[i] == max_element)
7               max_index = i
8       ENDFOR
9       removeLeaf(H, max_index)
10  enddef
11
12
13  DEF removeLeaf(Heap, index)
14      Heap[index] = Heap[Heap.size]
15      Heap.size = Heap.size-1
16
17      WHILE (index > 1)
18          parent = index/2
19          IF (Heap[index] < Heap[parent])
20              swap(Heap, index, parent)
21              index = parent
22          ELSE
23              BREAK
24      ENDWHILE
25  ENDDEF
26
27
28  DEF swap(Heap, index1, index2)
29      tmp = Heap[index1]
30      Heap[index1] = Heap[index2]
31      Heap[index2] = tmp
32  ENDDEF
```

In a min heap having n elements, there are $ceil(n/2)$ leaf nodes. Since lines 5, 6 and 7 each have a time cost of 1 and are repeated $ceil(n/2) - 1$ times, the time complexity of the FOR loop in DeleteMax taks $\Theta(n/2) = \Theta(n)$. When removing a leaf from the heap all the instruction inside the WHILE loop have a time cost of 1 and are repeated no more than h times where h is the height of the heap = $floor(log_2(n))$. So the time complexity to remove a leaf from the heap is: $O(log(n))$. In conclusion the time complexity to remove the maximum in the Heap is: $\Theta(n) + O(log(n)) = \Theta(n)$.

Exercise 1C

The worst case scenario for $DeleteMax$ on a heap H consisting in 8 nodes happen when the farthest leaf is smaller than the parent of the max value of the heap.

The procedure of $DeleteMax$ first find out which leaf is the max (the red one in the below example). Then swap the max node whit the farthest one (the green one in the below example) and decrease the size of the heap by one. After that if the green node is smaller than the parent of the max the heap priority is not respected so is necessary to swap this two nodes.



Exercise 2A

To evaluate the array B I wrote a Python code[3]. The execution of the program give as result B = [4, 0, 5, 3, 0, 0, 2, 0, 1, 0].

Exercise 2B

The pseudo-code to solve this problem work in this way. For each e element whit a index i in the input array counts the elements whit a greater index that are smaller than e and assign this value to the i-th value of the result array.

```
1  DEF exercice2b(A)
2      B = allocateArrayOfSize(A.Size)
3      FOR i=1 TO A.size
4          count = 0
5          FOR j=i+1 TO A.size
6              IF (A[j] < A[i])
7                  count += 1
8              ENDIF
9          ENDFOR
10         B[i] = count
11     ENDFOR
12     RETURN B
13 ENDDEF
```

Given n as the number of elements in the array. The time required to allocate an array of size n is: $O(n)$. Lines 6 and 7 have a cost of 1 each and are repeated $j \in [0, n - i]$ times by the inner FOR loop. Where $i \in [1, n]$. So the inner loop takes at most $n - 1$ repetition which give a cost of: $O(n)$. The external FOR loop over n values so take time of $O(n)$. Because the two loops are nested the total time complexity is $O(n) \cdot O(n) = O(n^2)$. In conclusion the time complexity of the algorithm is: $O(n) + O(n^2) = O(n^2)$.

Lets prove its correctness by induction. The base case is when A.size = 1. In this case the algorithm give as result an array of size 1 where the only value stored is 0 which is correct and follow the problem definition. Let suppose the algorithm works with an array A_n of size n, lets prove that the algorithm works also with an array A_{n+1} of size $n + 1$. Where $A_{n+1} = [A_n, v]$ and v is the new value added to the array. Let B_n, B_{n+1} be the two result arrays associated respectively to A_n and A_{n+1}. Then $B_{n+1}[n + 1] = 0$ and if $A[i] > v$ then $B_{n+1}[i] = B_n[i] + 1$ otherwise $B_{n+1}[i] = B_n[i]$ where $i \in [1, n]$ and those rules are correct and follow the problem definition.

Exercise 2C

In the case there are only a constant number of values in A different from 0. First we count all the negative values. Then by looping left to right all the array and updating how many negative values are remaining to visit we assign the correct i-th value for the output array when $A[i] = 0$. By taking track of the indexes of non zero value, we then need to calculate the value in the corresponding index of the output array using an analogue technique of the previous exercise.

```
1  DEF exercice2c(A)
2      B = allocateArrayOfSize(A.Size)
3      indexes_of_non_zero = emptyQueue()
4      negative_right_values = 0
5
6      // manage all the cases where A[i] = 0
7      FOR i=A.size DOWN TO 1
8          IF (A[i] == 0)
9              B[i] = negative_right_values
10         ENDIF
11         IF (A[i] < 0)
12             negative_right_values += 1
13         ENDIF
14         IF (A[i] != 0)
15             indexes_of_non_zero.enqueue(i)
16         ENDIF
17     ENDFOR
18
19     // manage the cases where A[i] != 0
20     WHILE (indexes_of_non_zero.size > 0)
21         count = 0
22         index = indexes_of_non_zero.dequeue()
23         FOR j=index+1 TO A.size
24             IF (A[j] < A[index])
25                 count += 1
26             ENDIF
27         ENDFOR
28         B[index] = count
29     ENDWHILE
30
31     RETURN B
32 EDNDEF
```

Given n as the number of elements in the array. The time required to allocate an array of size n is: $O(n)$. All the instructions inside the first FOR loop have a equivalent cost of 1. Since they are repeated n times this give a time cost of $\Theta(n)$.

Lines 24 and 25 have a cost of 1 each and are repeated $j \in [0, n - i]$ times by the second FOR loop. Where $i \in [1, n]$. So the second FOR loop takes at most $n - 1$ repetition which give a cost of: $O(n)$. Since we have guarantee that non zero value are constant, let be this number equal to c. The external WHILE loop over c values so take time of $\Theta(c)$. Because the two loops are nested the total time complexity is $\Theta(c) \cdot O(n) = c \cdot O(n)$.

In conclusion the time complexity of the algorithm is: $O(n) + \Theta(n) + c \cdot O(n) = \Theta(n)$

Lets prove its correctness by induction. The base case is when A.size = 1. In this case the algorithm give as result an array of size 1 where the only value stored is 0 which is correct and follow the problem definition. Let suppose the algorithm works with an array A_n of size n, lets prove that the algorithm works also with an array A_{n+1} of size $n + 1$. Where $A_{n+1} = [A_n, v]$ and v is the new value added to the array. Let B_n, B_{n+1} be the two result arrays associated respectively to A_n and A_{n+1}. Then $B_{n+1}[n + 1] = 0$ and if $A[i] > v$ then $B_{n+1}[i] = B_n[i] + 1$ otherwise $B_{n+1}[i] = B_n[i]$ where $i \in [1, n]$ and those rules are correct and follow the problem definition.

Exercise 3A

A red-black tree is a binary search tree which has the following red-black properties:

1. Every node is either red or black.
2. The tree's root is black.
3. All the leaves are black NIL nodes.
4. If a node is red, then both its children are black.
5. Every simple path from a node to a descendant leaf contains the same number of black nodes.

Exercise 3B

To calculate the height [4] of a Red Black Tree we need to find out the max distance from the root to a leaf. As in a Red Black Tree all leaf are black NIL node actually we need to find out the max distance from the root to a black leaf NIL node. Here we use a recursive approach searching for the max number of edges in a path from the root to a NIL node.

```
DEF RedBlackTreeHeight(T)
    IF (T.root == NIL)
        RETURN NIL
    ENDIF
    RETURN maxDistanceToNILLeaf(T.root)
ENDDEF

DEF maxDistanceToNILLeaf(node)
    IF (node == NIL)
        RETURN 0
    ELSE
        RETURN max(maxDistanceToNILLeaf(node.left),
                   maxDistanceToNILLeaf(node.right)) + 1
ENDDEF
```

Let n be the number of node in a Red Black Tree. Since we call the function maxDistanceToNILChild in both child of a node recursively starting from the root, we call this function n times meaning that the time complexity of the algorithm is: $\Theta(n)$

Since the maximum distance from the root to a leaf node is equivalent to the max number of edges in a path from the root to a NIL node. Then the algorithm correctly return the height of a Red Black Tree.

Exercise 3C

The black height of a Red Black Tree is the number of black nodes on any simple path from a node x (not including it) to a leaf. Since every path from a node to a descendant leaf contains the same number of black nodes to calculate the black height we can follow any path. In this pseudo-code I decided to take always the left path.

```
DEF RedBlackTreeBlackHeight(T)
    IF (T.root == NIL)
        RETURN NIL
    ENDIF
    RETURN countBlackNodesToNILLeaf(T.root)
ENDDEF

DEF countBlackNodesToNILLeaf(node)
    IF (node.left == NIL)
        RETURN 1
    ELSE
        IF (node.left.color == BLACK)
            RETURN countBlackNodesToNILLeaf(node.left) + 1
        ELSE
            RETURN countBlackNodesToNILLeaf(node.left)
        ENDIF
ENDDEF
```

Let n be the number of node in a Red Black Tree. Since we visit at most all the left descendants of the root we cal the function countBlackNodesToNILLeaf at most no more than h times where h is the height of the tree. Since h is $O(log_2(n))$ [5]. The time complexity of the alghorithm is $O(log_2(n))$.

Since in a Red Black Tree the number of black nodes in a path to a leaf node below a node x is equivalent to the number of edges that arrive in a black nodes in a path from x to a leaf node. Then the algorithm is correct.

Exercise 4A

To handle a single pair a proper data structure is a tuple of size two. A tuple is immutable, or unchangeable, ordered sequence of elements. In case your programming language do not support tuple a good alternative is an array of fixed size of two for each pair. In both case the first element stored in the data structure is always the a_i value and the second is the b_i value. The data structure to store the n pairs of integer values is an array.

To sort the n pairs of integer values a efficient algorithm is the Merge sort. Since we are ordering pairs we need to define a ordering criterium for pair. In this case I define the function isFirstPairMinor which is used to confront two pair and return true if the first one is minor false otherwise.

```
DEF mergeSort(array)
    IF (array.size > 1)
        mid = floor(array.size / 2)

        // Dividing the array elements into 2 halves
        L = allocateArrayOfSize(mid)
        R = allocateArrayOfSize(array.size-mid)
        FOR i=1 TO array.size
            IF (i<mid)
                L[i] = array[i]
            ELSE
                R[i-mid] = array[i]
            ENDIF
        ENDFOR

        // Sorting the first half
        mergeSort(L)
        // Sorting the second half
        mergeSort(R)

        i = j = k = 1

        // merge the two sorted array
        WHILE (i < L.size and j < R.size)
            IF (isFirstPairMinor(L[i], R[j]))
                array[k] = L[i]
                i += 1
            ELSE
                array[k] = R[j]
                j += 1
            ENDIF
            k += 1
        ENDWHILE

        // Checking if any element was left
        WHILE (i < L.size)
            array[k] = L[i]
            i += 1
            k += 1
        ENDWHILE

        WHILE (j < R.size)
            array[k] = R[j]
            j += 1
```

```
            k += 1
        ENDWHILE
    ENDIF
ENDDEF

DEF isFirstPairMinor(tuple1, tuple2)
    IF (tuple1[0] != tuple2[0])
        RETURN tuple1[0] < tuple2[0]
    ELSE
        return tuple1[1] <= tuple2[1]
    ENDIF
ENDDEF
```

For each call of mergeSort the function call itself twice with an array of half the initial size until there is the call of merge sort with an array of size one and in this case it simply return.

The time required to allocate an array of size $floor(n/2)$ is: $O(n)$. The time required to allocate an array of size $n - floor(n/2)$ is: $O(n)$. The inner content of the first FOR have a equivalent cost of 1 and is repeated n times which give a contribution of: $\Theta(n)$. All the other three WHILE have inner content that have a equivalent cost of 1 which is repeated at most no more than n times. This give a contribution of: $3 \cdot O(n)$. Let $T(n)$ the recursive function that describe the cost of mergeSort in relation to n. For the previous observations, an upper bound of his definition is:

$$T(n) \leq \begin{cases} 2 \cdot T\left(n/2\right) + 6 \cdot n & \text{if } n > 1 \\ 1 & \text{if } n \leq 1 \end{cases} \tag{1}$$

Since

$$T\left(\frac{n}{2}\right) \leq 2 \cdot T\left(\frac{n}{4}\right) + 6 \cdot \frac{n}{2}$$

If we replace the last equation inside (1) we find out that:

$$T(n) \leq 2 \cdot \left(2 \cdot T\left(\frac{n}{4}\right) + 6 \cdot \frac{n}{2}\right) + 6 \cdot n = 2^2 \cdot T\left(\frac{n}{2^2}\right) + 6 \cdot 2 \cdot n \tag{2}$$

Since

$$T\left(\frac{n}{4}\right) \leq 2 \cdot T\left(\frac{n}{8}\right) + 6 \cdot \frac{n}{4}$$

If we replace the last equation inside (2) we find out that:

$$T(n) \leq 2^2 \cdot \left(2 \cdot T\left(\frac{n}{8}\right) + 6 \cdot \frac{n}{4}\right) + 6 \cdot 2 \cdot n = 2^3 \cdot T\left(\frac{n}{2^3}\right) + 6 \cdot 3 \cdot n$$

By iterative this process i times we find out that:

$$T(n) \leq 2^i \cdot T\left(\frac{n}{2^i}\right) + 6 \cdot i \cdot n$$

Since there is the term $T\left(\frac{n}{2^i}\right)$ for sure after some iteration $\frac{n}{2^1} = 1 \implies n = 2^i$ and $log_2(n) = log_2(2^i) = i$

So

$$T(n) \leq n \cdot T(1) + 6 \cdot n \cdot log_2(n) = n + 6 \cdot n \cdot log_2(n)$$

Which means that $T(n) = O(n \cdot log(n))$

Exercise 4B

No there is no algorithm more efficient than the one proposed as solution of the previous exercise. This is because there are no constraint on the domain of the b_i's values.

Exercise 4C

In this case since $a_i \in [1, k]$ and $b_i \in [1, h]$ and both have limited values we can use Counting sort to solve the sorting problem in a more efficient way.

First we associate each pair (a_i, b_i) a v_i value such that $v_i = a_i \cdot k \cdot h + b_i \cdot min(h, k) = f(a_i, b_i)$. Then we sort all the v_i values using Counting sort. Let $v_1, ..., v_n$ they are lexicographically sorted if $v_i \leq v_{i+1}$ for all $i \in [1, n-1]$. After that we compute $f^{-1}(v_1), ..., f^{-1}(v_n)$ that give as result the lexicographically sorted pairs $(a_1, b_1), ..., (a_n, b_n)$ which are the final objective of this exercise.

Time Complexity analysis

First we need to calculate n v_i values starting from the pairs and this take time: $O(n)$. Then by knowing that Counting sort have a time complexity of $O(n + r)$. Where n is the number of elements in input array and r is the range of input. For the proof of the time complexity of Counting sort read reference [6]. Let $c = min(h, k)$. Since the max v value is: $k^2 \cdot h + h \cdot c$ and the minimum v value is: $k \cdot h + h$. The range of the v value is: $k^2 \cdot h + h \cdot c - k \cdot h - h + 1$. This means that time complexity required to sort n v_i values using Counting sort is: $O(n + k^2 \cdot h + h \cdot c - k \cdot h - h + 1)$. After that we compute $f^{-1}(v_1), ..., f^{-1}(v_n)$ that give as result $(a_1, b_1), ..., (a_n, b_n)$ and this takes time: $O(n)$. In conclusion the time complexity of the algorithm is: $2 \cdot O(n) + O(n + k^2 \cdot h + h \cdot c - k \cdot h - h)$.

Exercise 5A

Let n be the number of elements in the array and let m be the recursive median element of the chunks of size 5 of the array. During the lessons, we explicitly assumed that the input array does not contain duplicate values because this hypothesis is necessary for the proof of the time complexity. Without this hipotesis the upper bound for the number of elements smaller or equal to m became n tanks to the corner case when all the elements of the array are equal.

By relaxing this condition it may be the case that in each step of select we decrease the search space by only one and this like in Quick sort bring to a worst case scenario where the time complexity is: $O(n^2)$

Exercise 5B

To enhance the algorithm we have seen during the lessons I redefine the function partition and select. In this case partition create a three part partition. The limits of this parts are described by two variable fl (First Limit) and sl (Second Limit). The first part have positive indexes $\in [1, fl - 1]$ and contains elements of the array that are minor of the pivot. The second part have positive indexes $\in [fl, sl]$ and contain elements of the array that are equal to the pivot. The third part have positive indexes $\in [sl + 1, A.size]$ and contains elements of the array that are greater of the pivot. The select function is almost equivalent to what we defined during the lesson but have a stronger stopping criteria that handle better the case when the pivot's value have duplicates.

```

DEF partition(A, begin, end, pivot)
    swap(A[begin], A[pivot])
    fl = begin // will be the First Limit
    sl = end   // will be the Second Limit
    i = begin+1
    pivot = A[pivot]

    WHILE (i <= sl)
        IF (A[i] < pivot):
            swap(A[fl], A[i])
            fl += 1
            i += 1
        ELSE
            IF (A[i] > pivot)
                swap(A[i], A[sl])
                sl -= 1
            ELSE
                i += 1
            ENDIF
        ENDIF
    ENDWHILE
    RETURN [fl, sl]
ENDDEF

DEF select(A, begin, end, i)
    IF (end==begin)
        RETURN i
    ENDIF

    j = selectPivot(A, begin, end)
    array = partition(A, begin, end, j)
    fl = array[0]
    sl = array[1]

    IF (i<= sl and i >= fl)
        RETURN i
```

```
    ENDIF
    IF (i < fl)
        RETURN select(A, begin, fl-1, i)
    ELSE
        RETURN select(A, sl+1, end, i)
    ENDIF
ENDDEF
```

Let n the number of elements in the array. Let $T(n)$ the function that estimate the average time complexity of the select algorithm. Since selectPivot return an almost median element of the array and takes time $T(n/5)$. Since we estimate that the pivot selected is a median value then the recursive call of select inside select takes time: $T(n/2)$. Since the partition algorithm takes time n. Then for the previous description we can define :

$$T(n) \le \begin{cases} T\left(n/5\right) + T\left(n/2\right) + n^{\cdot} & \text{if } n > 1 \\ 1 & \text{if } n \le 1 \end{cases}$$

Supposing that $T(n)$ is $O(n)$ then $T(n_1) + T(n_2) \le T(n_3 + n_4)$ $\forall n_1, n_2, n_3, n_4 \in N$ such that $n_1 \le n_3$ and $n_2 \le n_4$. Then

$$T(n) \le \begin{cases} T\left(\frac{7 \cdot n}{10}\right) + n^{\cdot} & \text{if } n > 1 \\ 1 & \text{if } n \le 1 \end{cases} \tag{3}$$

Since

$$T\left(\frac{7 \cdot n}{10}\right) \le T\left(\frac{7^2 \cdot n}{10^2}\right) + \frac{7 \cdot n}{10}$$

If we replace the last equation inside (3) we find out that:

$$T(n) \le T\left(\frac{7^2 n}{10^2}\right) + n \cdot \left(1 + \frac{7}{10}\right) \tag{4}$$

Since

$$T\left(\frac{7^2 \cdot n}{10^2}\right) \le T\left(\frac{7^3 \cdot n}{10^3}\right) + \frac{7^2 \cdot n}{10^2}$$

If we replace the last equation inside (4) we find out that:

$$T(n) \le T\left(\frac{7^3 n}{10^3}\right) + n \cdot \left(1 + \frac{7}{10} + \frac{7^2}{10^2}\right)$$

By iterative this process i times we find out that:

$$T(n) \le T\left(\frac{7^i n}{10^i}\right) + n \cdot \sum_{j=0}^{j=i-1} \left(\frac{7}{10}\right)^j$$

Since

$$\sum_{j=0}^{j=i-1} \left(\frac{7}{10}\right)^j = \frac{\left(\frac{7}{10}\right)^i - 1}{\frac{7}{10} - 1} = \frac{10}{3} \cdot \left(1 - \left(\frac{7}{10}\right)^i\right)$$

Then

$$T(n) \leq T\left(\frac{7^i n}{10^i}\right) + \frac{10 \cdot n}{3} \cdot \left(1 - \left(\frac{7}{10}\right)^i\right)$$

Since there is the term $T\left(\frac{7^i n}{10^i}\right)$ for sure after some iteration $\frac{7^i \cdot n}{10^i} = 1 \Longrightarrow \left(\frac{7}{10}\right)^i = \frac{1}{n}$

So

$$T(n) \leq T(1) + \frac{10 \cdot n}{3} \cdot \left(1 - \frac{1}{n}\right) = 1 + \frac{10 \cdot n}{3} - \frac{10}{3}$$

Which means that $T(n) = O(n)$

References

[1] GeeksforGeeks: Maximum element in min heap
`https://www.geeksforgeeks.org/maximum-element-in-min-heap/`

[2] Dr. Shun Yan Cheung: Deleting a node (at a specific location) from a heap
`http://www.mathcs.emory.edu/ cheung/Courses/171/Syllabus/9-BinTree/heap-delete.html`

[3] Andrea Gonzato: Python code
`https://github.com/AndreaGonzato/AD-Homework-Batch-II/blob/main/code.py`

[4] NIST: Definition: The height of a tree
`https://xlinux.nist.gov/dads/HTML/height.html`

[5] CodesDope: Proof of height of red-black tree is $O(lg(n))$
`https://www.codesdope.com/course/data-structures-red-black-trees/`

[6] CodesDope: Counting Sort, section: Analysis of Counting Sort
`https://www.codesdope.com/course/algorithms-count-and-sort/`