Homework 2

April 30, 2021

0.1 Algorithmic Design Homework 2

0.1.1 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.

In case of a Min-Heap we know that $parent(p) \leq p$ so the maximum has to be found in the leaves of the Min-Heap which are $\lceil n/2 \rceil$ where n is the heap size. Considering the index starting from 0 we get that the first leaf will be at index $\lfloor n/2 \rfloor$.

The time complexity of this algorithm will be $\Theta(\lceil n/2 \rceil) = \Theta(n/2) = \Theta(n)$ because it requires to find the maximum among an unordered array of $\lceil n/2 \rceil$ elements.

[ ]:
```
# Pseudocode of exercise 1.a, I consider the index starting from 0

def find_max(A, l, r):
    # Finds the maximum value between [l,r) of an unordered array A
    current_max = A[l]
    i = l+1
    while i<r:
        if A[i] > current_max:
            current_max = A[i]
        i+=1

    return current_max

def RetrieveMax(H):
    return find_max(H, floor(H.size/2), H.size)
```

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

To delete the maximum element of an heap firstly we find the index corresponding to the maximum value, which takes time $\Theta(n)$, then we must delete this maximum element. This deletion is performed by storing the last element of H in a variable named `new_value`, then decreasing the size of the heap by 1, and finally we decrease the maximum value to `new_value` using the `DECREASE_KEY`

function seen during the lectures, which takes time $O(\log(n))$.
The overall time complexity of this operation is $\Theta(n)$ because finding the maximum takes time $\Theta(n)$ and decreasing a key takes time $O(\log(n))$.

[ ]:
```
# Pseudocode of exercise 1.b, I consider the index starting from 0

def find_max_index(A, l, r):
    # Finds the index of the maximum value between [l,r) of an unordered array A
    max_index = l
    current_max = A[max_index]

    while i<r:
        if A[i] > current_max:
            current_max = A[i]
            max_index = i
        i+=1

    return max_index

def DeleteMax(H):
    # Find the maximum index
    max_index = find_max_index(H, floor(H.size/2), H.size)

    # Delete the maximum
    # If the maximum index is the last index of H I simply delete it (best case)
    if max_index == H.size-1:
        H.size -= 1
        return

    # Otherwise I must decrease the maximum to the value of the last element in␣
↪H
    new_value = H[H.size-1]
    H.size -= 1
    DECREASE_KEY(H, max_index, new_value)
```

c) Provide a working example for the worst case scenario of the procedure `DeleteMax(H)` (see Exercise 1b) on a heap H consisting in 8 nodes and simulate the execution of the function itself.

In case of 8 nodes we will have $\lceil n/2 \rceil = 4$ leaves and the first leaf index will be at $\lfloor n/2 \rfloor = 4$, the algorithm first finds the index i that corresponds to the maximum value among $H[4], H[5], H[6], H[7]$ or more clearly $i = arg\,max_i H[i] \quad i \in \{4, 5, 6, 7\}$. Up to this point the algorithm is the same in the best and worst case but in the worst case we don't enter in the if block and proceed by creating the variable `new_value` which stores the value of the last element of the Min-Heap, then and decreases the size of the heap by 1 and finally the DECREASE_KEY function is called.
The DECREASE_KEY function in the worst case will swap the child and the parent up to the level before the root, it won't swap the root because the root is by definition the minimum value in the heap and so it must be less than or equal to `new_value` which was part of the heap.

So in our case we have $n = 8$ $h = 3$ and we only need one swap to fix the heap property. The picture below explains the deletion process:

0.1.2 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 $A[i+1, ..., n]$. More formally:

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

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

In this case we have $B = [4, 0, 5, 3, 0, 0, 2, 0, 1, 0]$

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.

The most straightforward method is to simply test the condition for each element of A, the pseudocode of this algorithm will be:

[ ]:
```
# Pseudocode of exercise 2.b, I consider the index starting from 1

def compute_B(A):
    n = A.size
    B = new array of size n

    for i = 1 upto n:
        count = 0
        for z = i+1 upto n:
            if A[z] < A[i]:
                count += 1
        B[i] = count

    return B
```

We can prove the correctness of this procedure by induction: * If A contains only one element then we don't enter the inner loop and $B = [0]$ as expected * Suppose that it works for A of n elements then we must prove that it works for $n + 1$
We must only prove that $B[1]$ contains the correct value since $B[2, ..., n]$ are correct by hypothesis, in this case $i = 1$ and the inner loop adds 1 to count each time $A[1] > A[z]$ for $z = 2, 3, ..., n \implies z = [2, n] \implies z = [i + 1, n]$ which is the definition of B for $i = 1$, so the procedure is correct

Finally the time complexity of this algorithm is given by the formula:

$$\sum_{i=1}^{n} \sum_{i+1}^{n} \Theta(1) = \alpha \sum_{i=1}^{n} (n - i) = \alpha \sum_{j=0}^{n-1} j = \frac{\alpha}{2} n(n-1) = \Theta(n^2)$$

Where I used $\Theta(1) = \alpha$ in and $j = n - i$

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.

Considering k values different from 0 we can construct one additional array to speed up the execution of the above algorithm, we will call this array the indexes array because it stores the indexes of the elements in A different from 0, this array can be computed in $\Theta(n)$ time and takes $\Theta(k)$ extra space. Below we write the pseudo-code of this algorithm

[ ]:
```
# Pseudocode of exercise 2.c, I consider the index starting from 1

def compute_B(A, k=NIL):
    n = A.size
    if k!=NIL:
        indexes = new array of size k
    else:
        indexes = a variable size container
    j = 1
    # Construct the indexes array
    for i = n downto 1:
        if A[i] != 0:
            indexes[j] = i
            j+=1

    k=indexes.size

    B = new array of size n
    n_zeros = 0 # Stores the number of zeros encountered so far
    for i = n downto 1:
        count = 0

        # Add n_zeros to count if A[i]>0
        if A[i] > 0:
            count += n_zeros

        # Go through the indexes
        for j=1 upto k:
            z = indexes[j]
            # If z is less or equal to i I break because z=i+1, ..., n
            # and z at the successive iteration will be less than z now
            if z <= i:
                break
            # Otherwise if A[i]<A[z] I increase count by 1
            if A[i] > A[z]:
                count += 1

        B[i] = count

        # If I find a zero update n_zeros
```

```
        if A[i] == 0:
            n_zeros += 1

    return B
```

To prove the correctness of this procedure we must first prove that at iteration i the variable `n_zeros` contains the number of zeros in $A[i+1, ..., n]$, this is true because we increase `n_zeros` by 1 iif $A[i] = 0$ and we do this at the end of the loop, so all the computations will use the value of `n_zeros` corresponding to one iteration before which will contain the zeros in $A[i+1, ..., n]$. Furthermore notice that the `indexes` array is a strictly decreasing array by construction

Now we can prove the correctness of the procedure * If A contains only one element we can split between two cases: $A[1] \neq 0$ and $A[1] = 0$ * If $A[1] \neq 0$ we enter in the indexes loop, however we suddenly exit it because $z = i = 1$ and so $B = [0]$ * If $A[1] = 0$ we don't enter the inner loop at all because the indexes array is empty and so $B = [0]$
We see that in both cases $B = [0]$ which is the correct result * Suppose that it works for A of n elements then we must prove that it works for $n + 1$

We must only prove that $B[1]$ contains the correct value since $B[2, ..., n]$ are correct by hypothesis, We split the proof in 2 cases: * If $A[1] > 0$ in this case we add `n_zeros` to count and then we proceed in the inner loop by checking only the values different from 0 until $z <= 1$ * If $A[1] \leq 0$ in this case we don't add `n_zeros` to count and then we proceed in the inner loop by checking only the values different from 0 until $z <= 1$
We see that this algorithm performs all the steps of the above one but it skips the checking of all the zeros which is done at the beginning by adding `n_zeros` or not, so it must be correct

The time complexity of this algorithm is the sum of two terms: 1. Creating the indexes array: $\Theta(n)$ because we must do a forward pass through A, the worst case will be when we don't know k, in that case we must use a container of variable size and we lose a bit of speed, however we still keep the $\Theta(n)$ complexity. 2. Executing the inner loop will take $\Theta(k)$ in the worst case (when all the elements different from 0 are at the end of A) and $\Theta(k/2) = \Theta(k)$ in the average case, since we have to execute this loop n times this will take $\Theta(nk)$

At the end we get a time complexity of $\Theta(nk)$ and an extra space requirement of $\Theta(k)$

0.1.3 Exercise 3

Let T be a Red-Black Tree

a) Give the definition of Red-Black Trees

A red-black tree is a binary search tree that satisfies the following properties:
1. Every node has a color associated with it which is either red or black.
2. The tree's root is black.
3. Every leaf is a black NIL node.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves 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 general we know that the height of a Red-Black Tree is bounded from above and below, in particular $\log(n+1) \leq h \leq 2\log(n+1)$, however we don't have a closed formula of the exact height of a Red-Black Tree. A possible solution to this problem can be found by noticing that if I know the height of the right h_r and left h_l child of a node x I can compute the height of x by simply computing $h(x) = \max(h_l, h_r) + 1 = \max(h(x.left), h(x.right)) + 1$, the last step gives us a recursive formula to compute the height of any node x and so even of the root of T, the base case of this recursion is when I reach a null node, in that case the height is set to 0 because I reached a leaf node whose height is by definition 0.

To prove the correctness of this algorithm we proceed by induction, suppose x is a leaf, the algorithm will return 0 which is the correct result. Now suppose it works for the left and right child of a node x, we must prove that it works for x, indeed we have $height(x) = \max(height(x.left), height(x.right)) + 1 = \max(h_l, h_r) + 1$ which is the correct result, we take the longest path and increase it by 1. As far as the complexity is concerned this algorithm will pass through all the nodes only once because each recursive step shifts the problem to the nodes one level below and will compute the same operation each time, so the complexity will be $\Theta(n)$ where n is the number of nodes in the tree.

[ ]:
```python
# Pseudocode of exercise 3.b

def height(x):
    if x==NIL:
        return 0

    return max(height(x.left), height(x.right)) + 1
```

Notice that in this algorithm we never used the color of the node, so it can also be used to find the height of a simple BST

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

The black-height of a node x is the number of black nodes below it, to compute it efficently we can simply travel down the tree until we reach a leaf while keeping track of the number of black nodes encountered so far, the correctness of this algorithm is given by property 5 of the Red-Black Tree data structure: it doesn't matter which simple path we choose because all the simple paths will always contain the same number of black nodes.
The time complexity of this algorithm will be $\Theta(h) = \Theta(\log(n))$ because we must execute the loop h times and the Red-Black Trees have the following property $\log(n+1) \leq h \leq 2\log(n+1)$

[ ]:
```python
# Pseudocode of exercise 3.c

def black_height(x):
    # By initializing the height to 1 I consider that the leaves are black nodes
    height = 1
    while x is not NIL:
        if x.color=="black":
```

```
            height += 1
        x = x.left

    return height
```

0.1.4 Exercise 4

Let $(a_1, b_1), ..., (a_n, b_n)$ 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;

The choice of the data structure depends on what we have to do with the pairs, if the pairs must be lexicographically sorted all the time and new pairs are added or removed frequently then a Red-Black Tree could be a good data structure, because it always keeps the pairs ordered and implements insertion, deletion and finding a pair in $\Theta(\log(n))$ time.

However if the pairs are not added or deleted after initialization a good data structure could be a simple array of pairs, if the values must be sorted or if we want to perform a find operation multiple times we could sort the values at initialization, this would take $\Theta(n \log(n))$ time using a general purpose sorting algorithm, which is the same as inserting n values in a Red-Black Tree. Furthermore this simple data structure saves space because we don't need to keep track of the pointers to left and right child and the color of the node. I will consider this data structure in the following exercises because sorting operation on a Red-Black Tree are never required since the tree is already sorted by default.

To order this array of pairs which from now on I will call it A I have chosen the merge sort algorithm because it orders a sequence in $\Theta(n \log(n))$ time and is a stable sorting algorithm.

[ ]:
```
# Pseudocode of exercise 4.a, I consider the index starting from 0

# Merge procedure
def merge(A, l, c, r, total_order):
    i, j, k = l, c, 0
    B = empty array of size r-l

    # Fill B with the smallest element between A[i] or A[j]
    while i<c and j<r:
        if total_order(A[i], A[j]):
            B[k] = A[i]
            i+=1
        else:
            B[k] = A[j]
            j+=1
        k+=1
```

```
    # Transfer the remaining left part if any
    while i < c:
        B[k] = A[i]
        i+=1
        k+=1

    # Transfer the remaining right part if any
    while j < r:
        B[k] = A[j]
        j+=1
        k+=1

    # Copy the content of B into A
    for k = l upto r-1:
        A[k] = B[k-l]


# Merge sort algorithm
def merge_sort(A, l, r, total_order):
    # We choosed to implement a vanilla merge sort, however here we can add
    # other base cases, for example we can perform an insertion sort if the
    # array size is less than a certain threshold, like in timsort

    if(r-l>1):
        c = floor((l+r)/2)
        merge_sort(A, l, c, total_order)
        merge_sort(A, c, r, total_order)

        merge(A, l, c, r, total_order)


# Lexicographical ordering of tuples of two numbers
def order(a, b):
    if a[0] < b[0]:
        return True
    if a[0] == b[0] and a[1] <= b[1]:
        return True

    return False
```

The merge procedure complexity is $\Theta(n)$, to compute the complexity of the whole merge sort we must solve the recursion:

$$T(n) = \begin{cases} \Theta(1) = \alpha & n \leq 1 \\ \Theta(n) + T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) & n \geq 1 \end{cases}$$

First of all we consider n power of two, then we generalize to non power of two, in this case we

have:

$$T(n) = \Theta(n)+2T(n/2) = \beta n+2(\beta n/2+2T(n/4)) = \alpha n+ \sum_{i=1}^{log_2(n)} \beta n = \alpha n+\beta n \log_2(n) = \Theta(n \log(n))$$

Where I used the equality $\Theta(n) = \beta n$.

In case of n not power of two we can use $T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) \leq 2T((n+1)/2) \leq 2T(cn/2)$ where $c < 2$ is a constant s.t. cn is a power of two, so we have

$$T(n) = \beta n+T(\lfloor n/2 \rfloor)+T(\lceil n/2 \rceil) \leq \beta n+2T(cn/2) = \beta n+2(\beta cn/2+2T(cn/4) \leq \alpha cn+ \sum_{i=1}^{log_2(cn)} \beta cn = \alpha cn+\beta cn \log_2(c$$

So the merge sort time complexity is the same for all n, a disadvantage of merge sort w.r.t. quicksort for example is the added space requirement which is $n/2 = \Theta(n)$, this space is required to store the array B in the merge phase

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.

A possible algorithm is to reorder A with respect to the first element of the tuple a_i which takes $\Theta(n + k)$ time with a counting sort algorithm. Then we must sort w.r.t. b_i, to do this we subdivide A in S subsequences such that each one will contain only tuples of the form $(c, *)$ where c is a constant and $*$ denotes any number, this operation takes $O(n)$ time, finally we sort each of these subsequences using the merge sort algorithm written above, this sorting will take $\sum_{j=1}^{S} \Theta(s_j \log(s_j)) = \Theta(\sum_{j=1}^{S} s_j \log(s_j))$ time where we indicated with s_j the length of the j-th subsequence, the pseudocode of this algorithm is written below.

However it's not clear if the algorithm is asimptotically faster than the one written above, to see if this is the case we restrict us to the case where $S = 2$, in this case $s_1 + s_2 = n$ and we have: $\Theta(n \log(n)) = \Theta((s_1+s_2) \log(s_1+s_2)) \geq \Theta(s_1 \log(s_1)+s_2 \log(s_2))$, so it seems that the algorithm is faster, the best case scenario is when $s_1 = s_2 = n/2$ which can be seen by minimizing the expression $s_1 \log(s_1) + (n - s_1) \log(n - s_1)$.

These results can be generalized to the sum above which leads $\Theta(\sum_{j=1}^{S} s_j \log(s_j)) \leq \Theta(\sum_{j=1}^{S} s_j \log(n)) = \Theta(n \log(n))$, and the minimum is found by minimizing $\sum_{j=1}^{S} s_j \log(s_j)$ with the constraint $\sum_{j=1}^{S} s_j = n$, this can be done via Lagrange multipliers where we get $s_j = n/S$, in this best case the complexity will be: $\Theta(n + k) + O(n) + \Theta(n \log(n/S)) = \Theta(k + n \log(n/S))$.

To summarize the complexity of this algorithm will be: * Counting sort on a_i: $\Theta(n+k)$ * Finding the first and last index of each subsequence of A: $O(n)$ * Sorting the subsequences A: $\sum_{j=0}^{S} \Theta(n_j \log(n_j)$

Overall cost: $\Theta(\sum_{j=1}^{S} s_j \log(s_j) + k)$
Best case: $\Theta(n \log(n/S) + k)$

This algorithm however is not asymptotically faster than the merge sort, to see this consider the best case and suppose that the counting sort is "free", that is, the elements are already ordered w.r.t. a_i, so we only need the merge sort phase which has a complexity $\Theta(n \log(n/S)) = \Theta(n \log(n) - n \log(S)) = \alpha n \log(n) - \beta n \log(S) = \Theta(n \log(n))$ so the asymptotic complexity is the same as the

merge sort and any comparison based sorting algorithms, this is quite curious because even if the merge phase is faster it is not asymptotically faster. To get a better algorithm we should use a non-comparison based sorting even for the b_i, however we don't have neither an upper bound to use counting sort, nor a distribution over b_i, which will allow us to use bucket sort.

Finally notice that if S increases as n increases s.t. n/S remains constant, then $\Theta(n\log(n/S)) \neq \Theta(n\log(n))$ and the merge phase will take time $\Theta(n\log(c)) = \Theta(n)$ so the overall algorithm will have a complexity $\Theta(n+k)$.

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.

We can apply two times the counting sort firstly on b_i then on a_i, like in the radix sort algorithm, the overall time complexity will be $\Theta(n+k) + \Theta(n+h) = \Theta(n+k+h)$

0.1.5 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?

To see why this assumption is necessary consider a pathological case: the input array A contains only one value repeated n times and suppose $n >> 1$ so that we don't enter in the base case, under these conditions we will go through the partition procedure, however at the end of it j will be either unchanged or equal to i, this means that one of the two partition arrays is empty and the other is full with n elements, so at the successive recursive call of `select` the input array will have size $n-1$ and the function will go through the same steps as before, calling another select with $|A| = n-2$ and so on, we can see that in the worst case we will have a time complexity of $\Theta(n^2)$ which is the result of $\sum_{i=0}^{n} n - i = 1/2n(n+1)$.

Notice that the algorithm will still find the correct index j, however we lose the linear time complexity

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

To enhance the algorithm we must change the `partition` procedure and in particular the case when $A[i] = A[p]$, an idea could be to resort to the case $A[i] > A[p]$ half the time and $A[i] < A[p]$ the other half, to do this we add a new variable $s \in \{0,1\}$ which tells us if we have to resort to the first case ($s = 0$) when $A[i] = A[p]$ or to the second one ($s = 1$), furthermore if we enter the first case then we will set $s = 1$ such that the successive time we encounter $A[i] = A[p]$ we will enter the second case and vice-versa. The pseudocode of this algorithm is shown below

[ ]:
```
# Pseudocode of exercise 5.b, I consider the index starting from 0

# Swappes element i and j of an array A
def swap(A, i, j):
    c = A[i]
```

```
    A[i] = A[j]
    A[j] = c


# Fixed partition procedure in case of duplicate values
def partition(A, i, j, p):
    swap(A, i, p)
    p = i
    i = i+1

    s = 0
    while i<=j:
        if A[i] > A[p]:
            swap(A, i, j)
            j-=1

        else if A[i] == A[p]:
            if s==1:
                # Execute the case A[i] > A[p]
                swap(A, i, j)
                j-=1
                # Make sure that afterwards we enter the case A[i] < A[p]
                s=0
            else:
                # Execute the case A[i] < A[p]
                i+=1
                # Make sure that afterwards we enter the case A[i] > A[p]
                s=1
        else:
            i+=1

    swap(A, p, j)
    return j


def select(A, i, l=0, r=NIL):
    if r==NIL:
        r=A.size - 1

    # Base case
    if r-l <= 10:
        SORT(A, l, r)
        return i

    # Recursive step
    p = SELECT_PIVOT(A, l, r)
    k = partition(A, l, r, p)
```

```
    if i==k:
        return k
    if i < k:
        return select(A, i, l, k-1)
    return select(A, i, k+1, r)
```

The complexity of this procedure will be $\Theta(n)$, to see this consider a pivot p, then d elements of A which I will call A_d will be different from $A[p]$ and e elements of of A I will call A_e will be equal to $A[p]$, we can split the overall partition in two cases:

- The partition of the A_d elements will work as before because we never enter the $A[i] = A[p]$ case and it will split the elements in S and G with an asymptotic ratio of $|S|/|A_d| \in [3/10, 7/10]$ if we use in the `SELECT_PIVOT` function chunks of size 5
- The partition for the A_e elements instead will split them "almost equally" between S and G, with almost equally I mean that one of the two array will have at most one equal element more than the other and so the asymptotic ratio will be $|S|/|A_e| \approx 1/2$.

With this we have demonstated that the partition will split A in two parts S and G with ratio $|S|/|A| = c \in [3/10, 7/10]$, so to get the complexity we must solve the recursion $T(n) = \Theta(n) + T(cn) = \alpha n + T(cn) = \sum_{i=0}^{k} \alpha n c^i$, which we bound from above and below: * $T(n) \leq \sum_{i=0}^{\infty} \alpha n c^i = L\alpha n$ where $L = 1/(c-1)$ is finite iif $c < |1|$ like in our case * $T(n) \geq \alpha n$ which is found by only keeping the first term of the sum

So $\alpha n \leq T(n) \leq L\alpha n \implies T(n) = \Theta(n)$