Homework Batch II: Trees and Algorithms

Algorithmic Design - A.A. 2020-2021

Valentina Blasone

Due date: 30/04/2021

Instructions

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 effciently return the maximum value in H without deleting it and evaluate its complexity.

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

(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.

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:

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

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

(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.

(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.

3. Let T be a Red-Black Tree.

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

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

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

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;

(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.

(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.

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?

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

Exercise 1

1. (a) The algorithm stems from the idea that the maximum cannot be located in a non-leaf node, because the min-heap property requires that the parent node is lesser than its children and hence a non-leaf node by definition has at least one child which has a larger value. Therefore we can limit our search to the leaf nodes, which in a min-heap are $\lceil n/2 \rceil$ (where n is the number of nodes). As a note, in all the exercise the array implementation of the min-heap covered in class is considered.

Algorithm 1 Return the maximum value in H without deleting it.

```
function RetrieveMax(H)
    n ← H.size
    max_value ← H[ CEIL(n/2) ]
    for i in CEIL(n/2) + 1 ... n:
        max_value ← max( max_value, H[ i ] )
    endfor
    return max_value
endfunction
```

Complexity

We indicate with n the size of min-heap H (i.e., the number of nodes).

- First assignments: $\Theta(1)$
- For each iteration of the for loop we have one comparison to find the maximum and one assignment: $\Theta(1)$. The loop is repeated $n - (\lceil n/2 \rceil + 1)$ times, therefore the complexity is:

$$O(n - (\lceil n/2 \rceil + 1))$$

Therefore the asymptotic complexity of the algorithm is $O(n)$.

1. (b) The first part of the algorithm is exaclty the same of the `retrieveMax` procedure developed in Exercise 1. (a), but for the fact that in this case both the maximum value and the corresponding position are returned. In a second step, the idea is to swap the node containing the maximum value, with the rightmost node in the tree and then delete it (in the array representation we are deleting the last element in the array). Finally, the `heapify` function seen during the lectures is used to recover the heap-property that may have been broken by the swapping.

Algorithm 2 Delete the maximum value from H.

```
function DeleteMax(H)
    # first we need to retrieve the position of the maximum
    n ← H.size
    max_value ← H [ CEIL(n/2) ]
    position ← CEIL(n/2)
    for i in CEIL(n/2)+1 ... n:
        if H [ i ] >= max_value then:
            position ← i
            max_value ← H [ i ]
        endif
    endfor
    H [ i ] ← H [ n ]
    H.size ← n - 1
    HEAPIFY(H, position)
endfunction
```

Complexity

- First two assignments: $\Theta(1)$
- For each iteration of the for loop, one boolean condition is checked plus at most two assignments: $\Theta(1)$

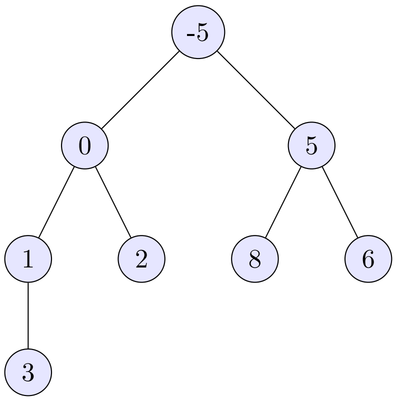
- Two more assignments: $\Theta(1)$
- HEAPIFY: $O(\,log\, n)$

The complexity of the algorithm is thus $O(\log\ n)$

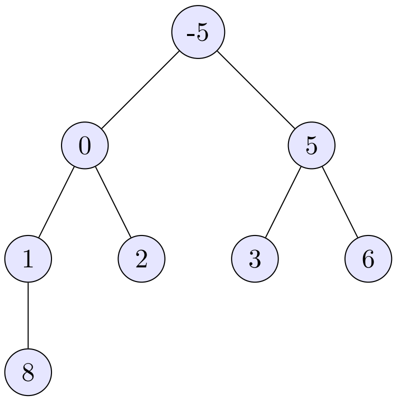
1. (c) Since the maximum value will always be a leaf, due to the heap property, the worst case scenario for the `deleteMax` algorithm is when the maximum is not in the lower level. To provide a working example we first build a random array containing 8 nodes, for which the worst case scenario applies:

$$A = [-5, 0, 5, 1, 2, 8, 6, 3]$$

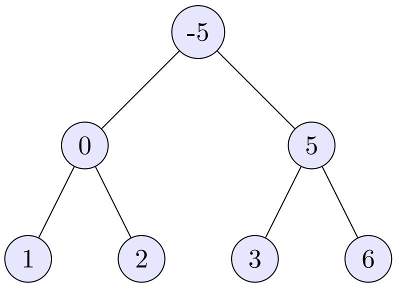
The corresponding heap is depicted below. In fact, the maximum value, 8, is in the second-last lower level.



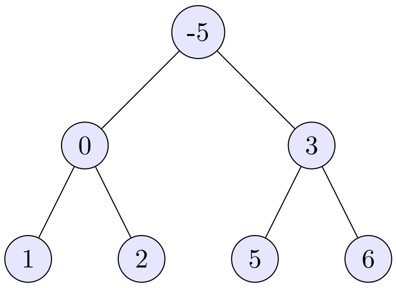
• The first step of the algorithm consists in swapping the maximum with the rightmost leaf in the lower level:



• The next step consists in deleting the rightmost node. The result is the following:



• Finally we need to fix the heap property. In this case it is sufficient to swap the node containing the value 3 in the last level with its parent. The result is shown in the following:



Exercise 2

2. (a) Given $A = [2, -7, 8, 3, -5, -5, 9, 1, 12, 4]$, the corresponding vector B is:

$$B = [4, 0, 5, 3, 0, 0, 2, 0, 1, 0]$$

2. (b) The algorithm is:

Algorithm 3 Compute the array B such that $B[\,i\,] = |\{z \in [\,i+1, n\,]\ s.t.\ A[\,z\,] < A[\,i\,]\}|$

```
function SMALLERVALUES(A)
    B ← ALLOCATE_ARRAY_OF_ZEROS(|A|)
    for i in 1 ... |A|:
        for z in i+1 ... |A|:
            if A[ z ] < A[ i ] then
                B[ i ] += 1
            endif
        endfor
    endfor
    return B
endfunction
```

Complexity

- Allocating and initializing B: $\Theta(n)$
- Two nested for loops:
$$\sum_{i=0}^{n}\sum_{j=i+1}^{n} 1 = \frac{1}{2}n(n+1) = O(n^2)$$

Therefore the asymptotic complexity of the algorithm is $O(n^2)$.

Correctness

- Ignoring the first assignment, we need to prove the correctness of the two nested for loops. We need to do it for any size of the array. We can identify the following invariant property that must hold for every loop iterations: at the end of the $z = j$ inner iteration of outer iteration i, it must hold:
$$B[\,\text{idx}\,] = \sum_{z=i}^{j} \begin{cases} 1 & \text{if} A[\,z\,] < A[\,\text{idx}\,] \\ 0 & \text{otherwise} \end{cases} \qquad \forall\ \text{idx} = 1, ..., i \text{ and } z = i, ..., j$$
- We now prove this invariant property by induction on the number of iterations.
  1. *Base Case*: before the loops start, we have (note that B.size == A.size):
  $$A = [k1, ..., kn]$$
  $$B = [0, ..., 0]$$
  $i = 1$ and $z = 1$ so $A[\,1\,] \not< A[\,1\,]$ and $B[\text{idx}]$ must be 0 for idx = 1. Indeed, $B[\,1\,] = 0$ and the invariant property holds.
  2. *Inductive step*: suppose $k > 1$ and that the invariant property holds for k-th iteration in the loop, i.e.,
  $$B[\,\text{idx}\,] = \sum_{z=i}^{k} \begin{cases} 1 & \text{if} A[\,z\,] < A[\,\text{idx}\,] \\ 0 & \text{otherwise} \end{cases} \qquad \forall\ \text{idx} = 1, ..., i \text{ and } z = i, ..., k \quad (*)$$
  We need to prove that the invariant property holds for $k+1$. We can observe that the values up to idx $= i-1$ are correct by (*). For the array position idx $= i$, we know that the algorithm is correct up to $z = j$. When $z = j+1$, either the value of B is position idx = i is increased by 1, if $A[\,z = j+1\,] < A[\,\text{idx}\,]$, or the value is kept constant if $A[\,z = j+1\,] \not< A[\,\text{idx}\,]$. In both cases the invariant property still holds. The same induction mechanism can be applied to the outer loop by simply reasoning on the "idx" index instead of on the "z" index and therefore the correctness of the algorithm is proved.

2. (c) The proposed algorithm exploits the assumption that the number of values different from zero is fixed, namely k, in order to reduce the complexity. We assume to always consider array of dimension larger than or equal to k. We introduce an auxiliary array C of size k to store dynamically the values different from zero found up to that point, plus a counter of the zero values, z, and a counter of the non-zero values, m, both considered up to that point. The idea is to loop in reverse order over the given array A and at each step i, if the value of $A[\,i\,]$ is different from zero, then m is increased by one and the value is saved in the array C in position m; if the value is equal to zero, the counter z is increased by one. Moreover, only if the value is larger then zero, the values of z is added to $B[\,i\,]$. Finally, there is an inner loop on the array C in which we increase $B[\,i\,]$ by one for every value of C larger than $A[\,i\,]$. Since we are looping on C only up to the number of non-zero values found up to now, we are sure that we are only considering the correct values in the comparison. Moreover, C has a fixed size k, independent on the length of the array and therefore the inner loop does not affect the asymptotic complexity. Finally, looping in reverse order allows to work in place, instead of first computing the auxiliary array of non-zero values and then building up B and the algorithm represents a good trade-off between time and space complexity.

Algorithm 4 Enhancement of Algorithm 3, considering A has exactly k values different from 0

```
function SMALLERVALUES_K(A,k)
    B ← ALLOCATE_ARRAY_OF_ZEROS(|A|)
    C ← ALLOCATE_ARRAY_OF_ZEROS(k)
    m ← 0 # counter for the number of values different from 0
    z ← 0 # counter for the number of values equal to 0
    for i in |A| ... 1:
        if A[ i ] > 0 then
            m += 1
            C[ m ] ← A[ i ]
            B[ i ] += z
        else if A[ i ] < 0 then
            m += 1
            C[ m ] ← A[ i ]
        else
            z += 1
        endif
        for j in 1 ... m:
            if C[ j ] < A[ i ] then
                B[ i ] += 1
            endif
        endfor
    endfor
    return B
endfunction
```

Complexity

- Allocating and initializing B: $\Theta(n)$
- Allocating and initializing idx: $\Theta(k)$
- Other two assignments: $\Theta(1)$
- For loop, for each iteration: $O(k)$ for the nested loop plus $\Theta(1)$ for the further operations

The overall asymptotic complexity of the algorithm is therefore:

$$\Theta(n) + \Theta(k) + \Theta(1) + O(n) \cdot (O(k) + \Theta(1))$$

Assuming k is a constant this simply becomes $O(n)$.

Correctness

As anticipated at the beginning of the algorithm, inside the for loop over the length of A, three different cases may be encountered:

- The value contained in $A[i]$ is strictly larger than zero: in this case we increase m by one (i.e., we increase the number of values different from zero found up to now), we place the value $A[\,i\,]$ in position m of the auxiliary array C and we sum to $B[i]$ the number of zeros encountered up to that point (i.e., the number of zeros that are in positions $> i$).
- The value contained in $A[\,i\,]$ is strictly smaller than zero: in this case we increase m by one, we place the value $A[\,i\,]$ in position m of the auxiliary array C.
- The value contained in $A[\,i\,]$ is equal to zero: this is the last possible case and we simply need to increase z by one (i.e., we increase the number of values equal to zero found up to now).

The only thing remaining in order for the algorithm to be correct, is to account for the number of values smaller than $A[\,i\,]$, among the values different from zero found up to now, which are stored in the first m positions of the auxiliary array C. This is done by the inner for loop, whose correctness follows directly from the demonstration in Exercise 1. (b).

Exercise 3

3. (a) A Red-Black Tree (RBT) is a Binary-Search Tree (BST) which for every node stores an additional attribute, the colour, and which satisfies some additional properties. More specifically:

- the colour of each node is either RED or BLACK;
- the tree's root is always BLACK;
- all the leaves are defined as BLACK `NIL` nodes;
- all the RED nodes must have BLACK children;
- all the branches starting from the same node must contain the same number of BLACK nodes and this must hold for every node in the RBT.

From the constraints on the nodes colour, it follows that the length of any branch from the root to the leaves is at most twice the length of any other and therefore RBT are approximately balanced.

3. (b) The algorithm is simply based on recursion.

Algorithm 5 Height of a RBT

```
function HEIGHTRBT(node)
    if node is NIL then:
        return 0
    endif
    node.h_left ← heightRBT(node.left_child)
    node.h_right ← heightRBT(node.right_child)
    node.h ← max(node.h_left, node.h_right) + 1
    return node.h
endfunction
```

Asymptotic complexity

- Each recursive call: first boolean condition is $\Theta(1)$ plus comparison to find the maximum is $\Theta(1)$.
- We repeat the recursive call for a number of times equal to the number of nodes in the three, n.

Therefore, the complexity is $\Theta(n) \cdot 2 \cdot \Theta(1)$ and the asymptotic complexity is $\Theta(n)$.

Correctness

We prove the correctness of the algorithm by induction.

- *Base Case:* the node is *NIL* (a leaf) and the algorithm returns 0, which is indeed the height of a leaf, hence it is correct.
- *Inductive Step:* assume the algorithm is correct up to k recursive steps, which means that at the k-th level, the `node.h_left` variables store the correct height of the left child of the nodes at level $k+1$ and the `node.h_right` variables store the correct height of the right child of the nodes at level $k+1$, respectively. At this point, the heights of the nodes at the $k+1$ level are simply retrieved as the maximum between the heights of their children, which is correct. By induction the algorithm works up to the root of the considered tree (or sub-tree), which is the node at the last level.

3. (c) The algorithm is implemented using recursion.

Algorithm 6 Black Height of a RBT.

```
function BLACKHEIGHTRBT(node)
    if node is NIL then:
        return 0
    endif
    child ← node.left_child
    child.bh ← blackHeightRBT(child)
    if child.colour == BLACK then
        node.bh += 1
    endif
    return node.bh
endfunction
```

Asymptotic Complexity

We observe that the complexity of the algorithm depends on the height of the RBT, therefore first we try to bound this value.

- First, we prove that a sub-tree rooted in x has at least $2^{BH(x)} - 1$ internal nodes. We can do this by using induction on the height of the tree. We consider an arbitrary node x:
  - If x is a leaf, the sub-tree has no internal nodes and, indeed, BH(x)=0 and thus $2^0 - 1 = 0$.
  - If x is not a leaf, it has at most two children, x_l and x_r, both with smaller height with respect to x. We assume that the result is true for h_l and h_r. We observe that:

    $$BH(x\_l) \leq BH(x) \text{ and } BH(x\_l) \geq BH(x) - 1$$

    $$BH(x\_r) \leq BH(x) \text{ and } BH(x\_r) \geq BH(x) - 1$$

    We can express the number of internal nodes of the sub-trees rooted in x_l and x_r in terms of BH(x), as follows:

    $$n(x\_l) \geq 2^{BH(x)-1} - 1$$
    $$n(x\_r) \geq 2^{BH(x)-1} - 1$$

    From which:

    $$n(x) \geq n(x\_l) + n(x\_r) = 2(2^{BH(x)-1} - 1)$$
    $$\Rightarrow n(x) \geq 2^{BH(x)} - 1$$

    Finally, let's assume x is the root of the tree.

- From the definition of RBT, since the leaves are BLACK and it is not possible to have two consecutive RED nodes, the number of RED nodes is at most equal to the number of BLACK nodes on any branch. Thus, at least half of the nodes on any branch from a root towards the leaves must be BLACK. Therefore we can bound the black height of x:

  $$BH(x) \geq h(x)/2$$

- From the points above, it follows that:

  $$n(x) \geq 2^{h(x)/2} - 1$$
  $$\log_2(n+1) \geq h(x)/2$$
  $$h(x) \leq 2\log_2(n+1)$$

  and therefore:

  $$h(x) \in O(\log_2 n)$$

Finally, we observe that for every recursive call the cost of the operations is $\Theta(1)$, therefore the running complexity of the algorithm is $O(\log_2 n) \cdot \Theta(1)$, with asymptotic complexity equal to $O(\log_2 n)$.

Correctness

We prove the correctness of the algorithm by induction.

- *Base Case:* the node is *NIL* (a leaf) and the algorithm returns 0, which is indeed the black height of a leaf, hence it is correct.
- *Inductive Step:* assume the algorithm is correct up to k recursive steps, which means that at the k-th level, the `child.bh` variable stores the correct black height of its left child (since all paths are equivalent in terms of black height we decide to always select the left child; this is just a design choice which does not affect the asymptotic complexity). At this point, if the child is BLACK, we add 1 to the black height of the considered node at level $k+1$. If the child is red, the black height of the node at level $k+1$ remains equal to the black height of its child, which is correct. By induction the algorithm works up to the root of the considered tree (or sub-tree), which is the node at the last level.

Exercise 4

4. (a) An efficient abstract data type to store the pairs is the array. More specifically, we can use an array of arrays, so that we have an array for each pair (e.g., $[a_i, b_i]$) and an array that stores all the arrays containing the pairs: A = [[a_1, b_1], ..., [a_n, b_n]]. To efficiently sort the pairs in lexicographical order we can use `heap_sort` (as we defined it during lectures), by passing the lexicographical order as total order. In particular, we must slightly modify the `build_heap` algorithm in order to generalize it to any given total order.

Algorithm 7 Lexicographical order.

```
function LEXICOGRAPHICAL_ORDER(p1,p2) # p1 and p2 are two arrays [a1,b1] and [a2,b2]
    if p1[1] != p2[1] then:
        return p1[1] ≤ p2[1]
    else
        return p1[2] ≤ p2[2]
    endif
endfunction
```

Algorithm 8 Lexicographical sorting.

```
function LEXICOGRAPHICAL_SORTING(A)
    HEAP_SORT( A, total_order = lexicographical_order )
endfunction
```

Complexity

The complexity of the algorithm is the one of heap sort, thus $O(n \log n)$. Note that due to the total order used, in the worst case scenario we have to perform two comparisons for each pairs comparison. Nonetheless, this does not affect the asymptotic complexity.

4. (b) We assume that it exists a natural value k constant with respect to n, such that $a_i \in [1, k]$ for all $i \in [1, n]$. If we could assume that all the values of a_i are distinct, then the sorting will exclusively be based on a_i and therefore we could exploit the fact that the domain of a_i is bounded to use an algorithm not based on comparisons, that sorts in linear time. However, in the general case we cannot make this assumption and therefore the answer to the question is no, since the complexity lower bound for the algorithms based on comparisons is $\Omega(n \log n)$.

4. (c) We assume that the hypothesis on a_i of Exercise 4. (b) holds still and in addition we assume that it exists a natural value h, constant with respect to n, such that $b_i \in [1, h]$ for all $i \in [1, n]$. In this case, we can exploit the fact that all the values contained in the array are bounded, in order to use an algorithm not based on comparisons, able to sort values in linear time. For instance, we can use radix sort, where the values a_i and b_i are respectively the first and the second "digits" in the terminology that we have seen during lectures. The algorithm works by first of all sorting the pairs with reference to the second "digit" and then according to the first "digit". The sorting is performed by using a stable algorithm (i.e., that preserves the relative order, like counting sort). The complexity of the algorithm of course depends on the algorithm that we use to sort the

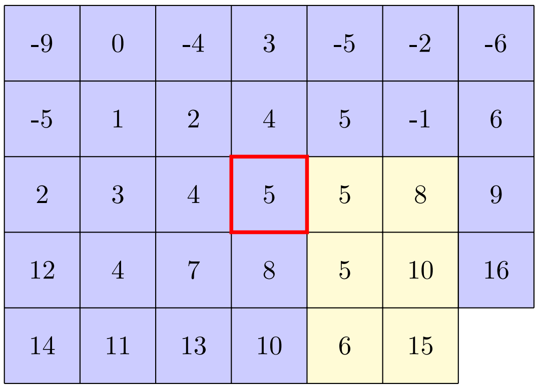
digits, so if the digit sorting takes time $\Theta(|A| + k)$, then radix sort takes time $\Theta(2(|A| + k))$, since we have two "digits" in each of A's values. Since the number of "digits" is upper bounded we have a linear algorithm in the number of "digits" we are dealing with.

Exercise 5

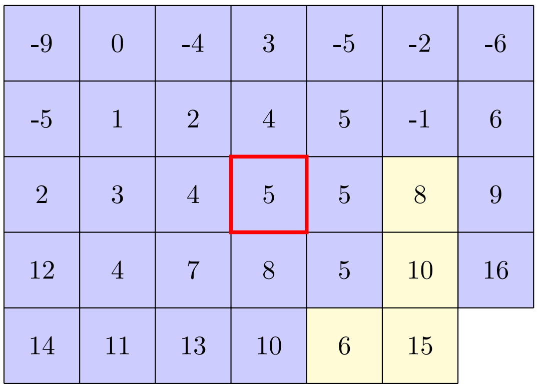
5. (a) The assumption of having all distinct values allowed us to reduce the complexity of the algorithm. In case of duplicates, it may not be possible to get a good split if the median value has many duplicates, resulting in slowing down the algorithm. In fact, whenever we are not able to balance enough the length of the arrays S and G we fall into a very large number of operations, $O(n^2)$.

More specifically, if we consider the `select_pivot` algorithm, the assumption that all values are distinct allowed us to say that at last one half of the medians, found by dividing the original array in chunks of five elements, are greater than or equal to the median-of-medians (which is found by recursively call `select` on the array of the medians). This allowed us to define an upper bound on the number of elements that smaller or equal than the partitioning element, and therefore an upper bound on the number of times in which `select` is called on the original array. If the assumption of distinct values does not hold anymore, in principle we may exceed this upper bound. We can see in a practical example how the reasoning we did in class may go wrong.

In fact, in deriving an upper bound for the number of values smaller or equal than the median-of-medians we ended up in identifying the cells that for sure and at least contain values strictly grater than the median-of-medians. This region is shaded in yellow in the picture on the right, which shows the chuncks as columns of a matrix, as we did in the lectures. This consideration allowed us to bound the maximum number of cells containing values smaller or equal than the median-of-medians. However, in case of duplicates like in this example, we notice that the actual number of cells containing values strictly larger than the median-of-medians is smaller, as indicated in the picture on the right. As a consequence, the upper bound that we found for the number of cells whose value is smaller than or equal to the median-of-the medians may be exceeded.

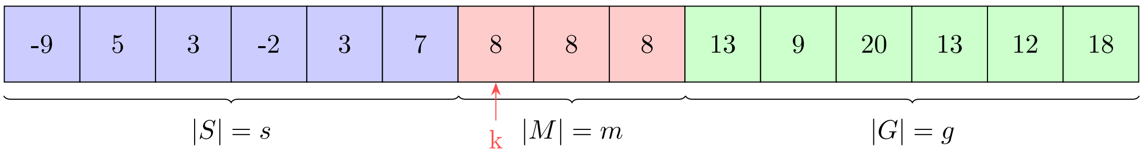


-9	0	-4	3	-5	-2	-6
-5	1	2	4	5	-1	6
2	3	4	5	5	8	9
12	4	7	8	5	10	16
14	11	13	10	6	15	



-9	0	-4	3	-5	-2	-6
-5	1	2	4	5	-1	6
2	3	4	5	5	8	9
12	4	7	8	5	10	16
14	11	13	10	6	15	

5. (b) The idea is to modify the partitioning phase, by introducing a third partition, in which we store all the values equal to the pivot. The partition is placed in the middle of the array. For clarity, we call $\overline{A}$ the array A after the computation; we call S the left partition, which stores the values strictly smaller than the pivot value and has size s; we call G the right partition, which stores the values strictly larger than the pivot value and has size g; finally we call M the middle partition, which stores the pivot value and all its duplicates and has size m. We call i the target index and k the index representing the first occurrence of the pivot value in the array after the calculation. An illustrative example of how the array is partitioned is shown in the following, assuming the pivot value is 8.



-9	5	3	-2	3	7	8	8	8	13	9	20	13	12	18

After calling the updated partition algorithm, in the updated select algorithm, we have the following possibilities:

- if $k \leq i < k + m$ we can return k
- if $i < k$ then $\overline{A}[\,i\,]$ is in S
- if $i \geq k + m$ then $\overline{A}[\,i\,]$ is in G

To enhance the algorithm seen during lessons in order to better deal with duplicate values, we first need to modify the `partition` and `select` algorithms into the updated `partition_dupl` and `select_dupl` agorithms. Moreover, in the `select_pivot` algorithm, `select_dupl` should also be used instead of `select`.

Algorithm 9 Partition with duplicates.

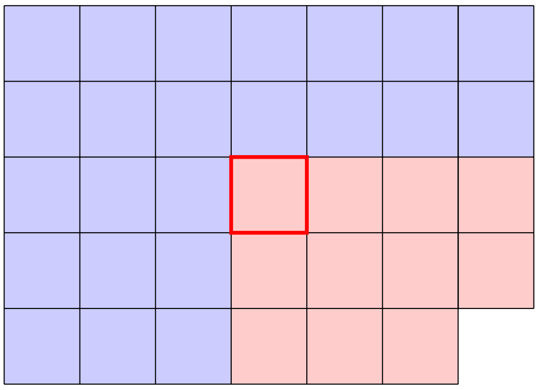
```
function PARTITION_DUPL(A,i,j,p)
    swap( A, i, p )
    ( p, i ) ← ( i, i+1 )
    p_last ← p + 1 # additional variable that stores the position for the next pivot duplicate
    while i ≤ j:
        if A[ i ] > A[ p ] then # if A[ i ] is greater than the pivot
            swap( A, i, j ) # place it in G
            j ← j - 1 # increase G size
        else if A[ i ] < A[ p ] then # if A[ i ] is smaller than the pivot, it is already in S
            i ← i + 1
        else # we found a duplicate of the pivot
            swap( A, i, p_last ) # we place the value in the next pivot duplicate position
            p_last ← p_last # we increase the next pivot duplicate position
        endif
    endwhile
    for p̂ in p ... p_last: # place all the pivot values in M, between S and G
        swap( A, p̂, j )
        j ← j - 1
    endfor
    m ← p_last - p # store the number of duplicates
    return j + 1, m # j contains the first pivot position at this point
endfunction
```

Algorithm 10 Select with duplicates.

```
function SELECT_DUPL(A,l=1, r=|A|, i)
    if r - l ≤ 10 then # base case
        SORT(A, l, r)
        return i
    endif
    j ← SELECT_PIVOT( A, l, r )
    k ← PARTITION_DUPL( A, l, r, j )
    if k ≤ i < k + m then
        return k
    else if i < k then # search in S
        return SELECT( A, l, k-1, i )
    else # search in G
        return SELECT( A, k+m, r, i )
    endif
endfunction
```

Complexity

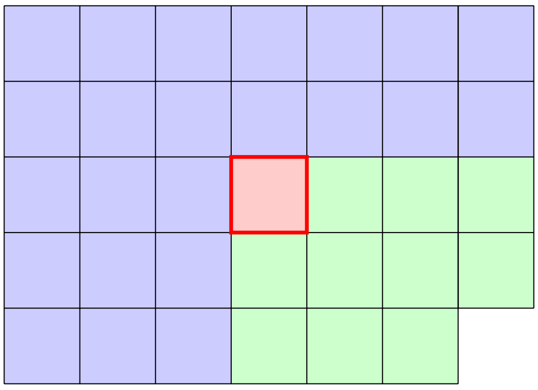
We can follow a procedure similar to the one we have seen during lectures. However we consider the case in which in the original array of size n we have m duplicates of the medians-of medians value. If we proceeded exactly as we did in class, we would have found that the number of elements larger or equal than the median-of-medians cannot be bounded, in fact in the worst case we may encounter the situation depicted in the picture below, where the blue cells indicate elements strictly smaller than the median-of-medians and the red cells represent the median-of-medians and its possible duplicates.



However, the fact that we use a three-partitioning algorithm solves this issue, in fact given the medians-of-medians as pivot, only the blue elements will go in the array S, whereas the red elements will go in the array M. In this extreme case the array G will be empty. The aforementioned pivot can be identified as the first element of the M array. At this point, either the desired index falls inside M and the algorithm terminates, or we are left with the array S, whose size is bounded. In a more general case, all the elements non-blue will go either in M if they are equal to the pivot, or in G, if they are strictly larger than the pivot.

This was just an example, but we can proceed with a more formal proof. The key point is that using the modified version of the partition algorithm we don't have to bound anymore the number of values smaller or equal than the median-of-medians, but it is enough that we bound the number of elements strictly smaller than the median-of-medians, i.e., the cells shaded in blue in the following picture. The values in green can be in the general case equal to the value of the median-of-medians, but not smaller. This number corresponds to:

$$\left\lfloor \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rfloor \cdot 5 + \left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil \cdot 2 \leq \left(\frac{n}{10}+1\right)\cdot 5 + \left(\frac{n}{10}\right)\cdot 2 = \frac{7n}{10}+5$$



The same reasoning applies to find an upper bound for the number of elements strictly larger than the pivot.

We can finally compute the complexity for the worst case scenario of the `sleect_dupl` algorithm.

- The first operations on the original array to divide it is groups of 5 elements each and at most one containing the remaining n mod 5 elements has complexity $\Theta(n)$.
- The operation of sorting the elements of each group and finding the median for each chunk and build an array containing all the median has complexity again $\Theta(n)$.

- The recursive calls to find the median-of-the medians have complexity $T\left(\left\lceil \frac{n}{5} \right\rceil\right)$.
- `partition_dupl` is called on the input array with the median-of-medians as pivot. If d is the number of distinct values in the array, then it is called at most d times in the `select_dupl` algorithm: $\Theta(d) \in \Theta(n)$
- If $k \leq i < k + m$, k is returned, otherwise `select_dupl` is called recursively on either S or G, but in any case it takes time at most $T\left(\frac{7n}{10} + 5\right)$.

Therefore:

$$T_S(n) = T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7n}{10} + 5\right) + \Theta(n)$$

We can proceed with the computation of $T_S(n)$ by substitution method exactly as we did in class, with the only difference in the constant ”+5” instead of ”+6”. Therefore the asymptotic complexity of the modified algorithm is again $O(n)$.