University of Trieste

International School for Advanced Studies

The Abdus Salam International Centre for Theoretical Physics

Algorithmic Design

Homework 2

Author:
Marco Sciorilli

Friday 30th April, 2021

Where not otherwise specified, the functions used in the pseudo-codes not explicitly written down are defined as the one seen during lectures, explained in the file `Algorithmic_Design_Notes.pdf` in this same repository.

Exercise 1

a.

Let's assume we are working on Binary Heaps, in an array implementation. As the last node has index n, its parent has index $n/2$, so all the leaves has indexes from $n/2$ to n. As for the structure of the min-heap the maximum value in H is stored in a leaf, we can write the following pseudo-code:

Algorithm 1: Find maximum in an array

```
def LASTMAX(A, l, r):
    j←A[l]
    for i in l..r do
        if j<A[i] then
            j← A[i]
        end
    end
    return j
enddef
```

Algorithm 2: Array-based representation: find maximum

```
def RetrieveMax(H):
    return FINDMAX(H, PARENT(|H|)+1, |H|)
enddef
```

`FINDMAX` takes as inputs and array A, a starting point index l, end an endpoint index r. It performs a pairwise comparison of the elements of the array from index l to index r, storing on the fly the greatest value found on the variable j (like a single iteration of `Selection sort`). As swap takes time $\Theta(1)$, the total cost of `FINDMAX` is $O(r-l)$. As `RetriveMax` only calls `FINDMAX`, its complexity is at most $O(n-n/2) = O(n/2) = O(n)$.

b.

Algorithm 3: Swapping maximum in a section of an array with the last value of the section, and returning the index of the other node

```
def SWAPMAX(A, l, r):
    j←l
    for i in l..r do
        if j<A[i] then
            j← i
        end
    end
    swap(A, j, r)
    return j
enddef
```

Algorithm 4: Deleting maximum in a Heap H

```
def DeleteMax(H):
    j←SWAPMAX(H, PARENT(|H|)+1, |H|)
    H.size ← H.size-1
    if H[PARENT(j)]⪯H[j] then
        error(H["+j+"]"is not smaller than its parent)
    end
    while not (IS_ROOT(j) or H[PARENT(j)] ⪯ H[j]) do
        swap(H, j, PARENT(j))
        j← PARENT(j)
    end
enddef
```

Using `SWAPMAX` swap the maximum value with the one on the rightmost leaves. This takes linear time, as the algorithm has the same complexity of `LASTMAX` +1 to account for the swap, so at most $O(n/2 + 1) = O(n)$. Deleting the now-rightmost-leaf, the maximum, takes constant time, and is not problematic for the preservation of the heap-properties. The swapped node on the other end could now be smaller than its parent. To account for that, we can follow the same procedure as for the algorithm `DECREASE_KEY`, progressively swapping the child with its parent until the heap-properties are respected, or until we reach the first common ancestor(as surely both of them are greater than it). Since the common ancestor can be the root, and the heap height is $\lfloor \log_2(n) \rfloor$, the complexity of this part is $O(\log n)$.
As $\log n \in O(n)$, the worst case scenario takes time $O(n)$.

c.

The worst case scenario happens when all the leaves have values in increasing order except for the last one, who is an only child, and after the swap the new child has to climb up to the last-most node before the root to fix the heap properties.

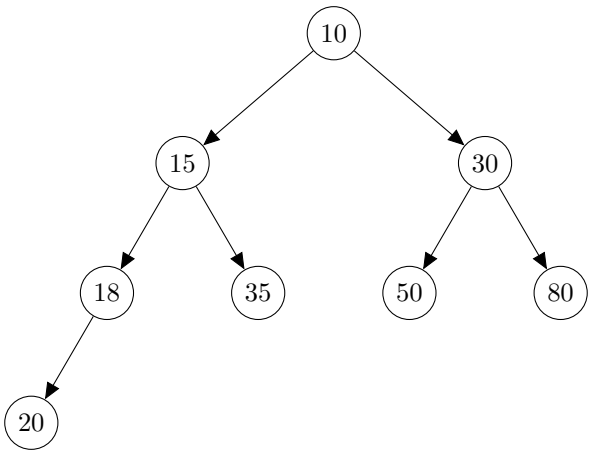


Figure 1: Worst case scenario heap

In the array representation

10	15	30	18	35	50	80	20

In out case PARENT($|H|$) + 1 = 5, so `SWAPMAX` does

10	15	30	18	35	50	80	20

↓

35

10	15	30	18	35	50	80	20

↓

50

10	15	30	18	35	50	80	20

↓

80

→

10	15	30	18	35	50	80	20

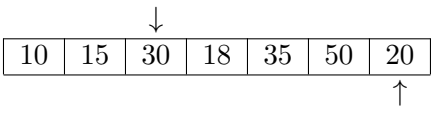
←

10	15	30	18	35	50	20	80

Now the rest of `DeleteMax(H)`

10	15	30	18	35	50	20	80

10	15	30	18	35	50	20



10	15	20	18	35	50	30

And we have the required Heap with the maximum deleted:

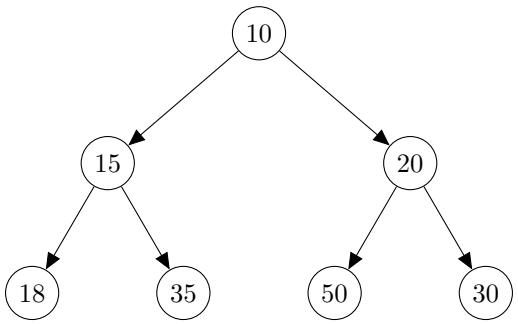


Figure 2: Worst case scenario heap, maximum removed

Exercise 2

a.

A is given

$$A = [2, -7, 8, 3, -5, -5, 9, 1, 12, 4] \tag{1}$$

So:

- $B[1] = |[-7, -5, -5, 1]| = 4$
- $B[2] = |[\emptyset]| = 0$
- $B[3] = |[3, -5, -5, 1, 4]| = 5$
- $B[4] = |[-5, -5, 1]| = 3$
- $B[5] = |[\emptyset]| = 0$
- $B[6] = |[\emptyset]| = 0$
- $B[7] = |[1, 4]| = 2$
- $B[8] = |[\emptyset]| = 0$

- $B[9] = |[4]| = 1$
- $B[10] = |[\emptyset]| = 0$

So B is

$$B = [4, 0, 5, 3, 0, 0, 2, 0, 1, 0] \tag{2}$$

b.

Pseudo-code

Algorithm 5: COUNTLESS: receives as inputs and array and an index, returns the number of elements of the array stored after the index which are smaller than the one stored at the index itself.

```
def COUNTLESS(A, l):
    j←0
    for i in l+1..|A| do
        if A[i]<A[l] then
            j← j+1
        end
    end
    return j
enddef
```

Algorithm 6: LESSARRAY: receives as inputs and array, returns an array (let's call him B) of the same size s.t. $B[i] = |\{z \in [i+1, n] | A[z] < A[i]\}|$

```
def LESSARRAY(A):
    B← ALLOCATE_ARRAY(|A|, default value = 0)
    for i in 1..|A| do
        B[i]← COUNTLESS(A, i)
    end
    return B
enddef
```

Complexity of the algorithm

Given n the length of the input array A, algorithm `LESSARRAY` calls `COUNTLESS` n times. At every call $k = [1, ..., n]$, the function `COUNTLESS` performs at most $n - k$ operations (in the case the elements of A are in decreasing order). So the time complexity of the algorithms is, using the gauss trick for summation

$$\Theta\Big(\sum_{i=1}^{n} n - i\Big) = \Theta\Big(\sum_{i=1}^{n-1} i\Big) = \Theta\Big(\frac{n}{2}(n+1) - n\Big) = \Theta(n^2) \tag{3}$$

Correctness of `COUNTLESS`

Before the iteration: An array A of cardinality $|A|$, where elements k s.t. if $k \in [A[l], ..., A[|A|]]$ then $k \in \mathbb{Z}$.

After the iteration: A is left unchanged. For all elements k touched by the algorithm, it is always possible to determine an order relation $\preceq$, in this case $\leq$. So every time the order relation is satisfied, the integer counter j is increased by 1. The output of the algorithm after the iteration is always the number of elements of the array from index l to the end which respect the order relation relative to the element stored in $A[l]$.

Correctness of `LESSARRAY`

Before the iteration: An array A of cardinality $|A|$, where elements k s.t. if $k \in A$ then $k \in \mathbb{Z}$.
After the iteration: A is left unchanged. A new array B of the same dimension of A is created, and all its entries initialized to 0. To all elements of B is than assigned the value of the output of the function `LESSARRAY`, applied on the array A, on the element whose index is the same of the one interested by the current iteration of the algorithm on B. The output is than: $B[i] = |\{z \in [i+1, n] | A[z] < A[i]\}|$.

c.

Pseudo-code

Algorithm 7: CONSTCOUNT: receives as input an array and a value k, and returns an array of the indexes of the elements of A which are $\neq k$

```
def CONSTCOUNT(A, k):
    B← ALLOCATE_ARRAY(∅)
    j←1
    for i in 1..|A| do
        if A[i]≠ k then
            B.size←B.size+1
            B[j]←i
            j←j+1
        end
    end
    return B
enddef
```

Algorithm 8: LESSARRAY_EFFICIENT: receives as inputs and array, returns an array (let's call him B) of the same size s.t. $B[i] = |\{z \in [i+1, n] | A[z] < A[i]\}|$

```
def LESSARRAY_EFFICIENT(A):
    // initializations
    k←0
    chunkend← |A|
    knumber←0
    B← ALLOCATE_ARRAY(|A|, default value = 0)
    C← CONSTCOUNT(A, k)
    // for every element of the array C, starting from the bottom
     for i in |C|..1 do
        // for every element of the array A between two non-0 values
        for j in chunkend..C[i] do
            // if the value in A is more than k, add to the corresponding
            index in B the number of values k after the index
            if A[j]> k then
                B[j]←knumber
            end
            // if the element in A is greater than the non-k values after
            it add 1 to the corresponding value in B
            for l in i..|C| do
                if A[C[l]]< A[j] then
                    B[j]←B[j]+1
                end
            end
            // increase the number of k as going backward trough A
            knumber←knumber+1
        end
        // reduce the number of k by 1 to account for the non k value
        knumber←knumber-1
        // redefine the end of the chunk to go through the next one
        chunkend←j-1
    end
    return B
enddef
```

Complexity of the algorithm

`CONSTCOUNT` has time complexity $\Theta(n)$ where n is the length of the input array (as the variable initialization, and the operations in the if statement take constant time).
We can compute `LESSARRAY_EFFICIENT` complexity by taking into account:

- The initialisation of k ad B takes constant time.
- The call of `CONSTCOUNT` takes time $\Theta(n)$.
- The for loop:

- The for loops over i and j are complementary, together they go through 1 time every element of the array A. What's inside the two loops than is repeted a total of n times.
- Inside the loop over j, there are two if instances, which takes constant time (for a total cost of $\Theta(n)$) and another inner loop.
- The inner-most loop has a length increasing with i, with a max length of $|C|$

Considering the inner-most loop as always of the max length ($k = |C|$, which never happens even in the worst case scenario), we can find an upper bound for the algorithm complexity: $O(kn) \geq O$ of our algorithm.

So the complexity of the program depends on $k = const$.
If k is increasing with n, the complexity is $\Omega(n)$ in the best scenario, and $O(kn)$ in the worst case scenario (i.e. when $k = n$, so $O(n^2)$, as we are in the same case as of **b.**).
If however we keep k constant and increase n, in the evaluation of the complexity k becomes less and less relevant as $n \to \infty$, so the worst and case scenario tends to coincide, an the complexity of the algorithm is $\Theta(n)$.

Correctness of `CONSTCOUNT`

Before the iteration: An array A of cardinality $|A|$, with elements l s.t. if $l \in A$ then $l \in \mathbb{Z}$, and a integer $k \in \mathbb{Z}$
After the iteration: The array A is left unchanged. An empty array B is initialized as well as a counter j. For every element of the array A, if the element of the array is not equal to the integer number k, the dimension of B is increased by 1, and to the new element of B is assigned the value of the index of the element in A. As both the values of A and k are integers, the comparison is always possible. The algorithm eventually return the array B.

Correctness of `LESSARRAY_EFFICIENT`

Before the iteration: An array A of cardinality $|A|$, with elements l s.t. if $l \in A$ then $l \in \mathbb{Z}$.
After the iteration: A is left unchanged. A new array B of the same dimension of A is created, and all its entries initialized to 0. A third array C is initialized with the output of the function `CONSTCOUNT`, giving it the array A and 0 as input.
The algorithm pass through all the elements C (so the all non-0 values of A), starting from the end. As all the entries of A between two values of C are just 0, the algorithm go through all of this 0-chunks in A, adding 1 to a counter of 0s, and initializing the corresponding element in B with the counter (if the element with the same index in A is greater than 0, of course). After that, a for loop check if the value in A is greater than any of the non-0 elements with greater index (going through C, as their index is stored there).
Eventually, we are left with our complete B.
If there are no non-0 values, C has size 0, so all values of A are 0s, and B is left as it is, made correctly of 0s, as no element is smaller than a previous one.
If there are no 0 values, the size of C is the same of A, every chunk is of size 1, and the

code reduce to the one of **b.**, just going backward through A. The result is then again the correct B.

Exercise 3

a.

Definition 1. *RBTs are BSTs satisfying the following conditions:*

- *Each node is either a RED or 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 x (down to a leaf) contain the same number of black nodes.*

b.

Algorithm 9: NODEHEIGHT: receives as inputs a node, returns the number of levels below it.

```
def NODEHEIGHT(x):
    if x=NIL then
        return
    end
    return Max(NODEHEIGHT(x.left), NODEHEIGHT(x.right))+1;
enddef
```

Algorithm 10: RBTHEIGHT: receives as inputs a tree, returns its height.

```
def RBTHEIGHT(T):
    x← T.root
    return NODEHEIGHT(x);
enddef
```

Correctness of `NODEHEIGHT`

Before the iteration: A node n of an RBT.
After the iteration: The function works by recursively calling itself on both the children of the considered node, until it reaches the `NIL` leaves. At every recursive call, only the longest branches is passed on upward, adding a +1 to account for the current node. When a `NIL` node is reached, it returns. As only the longest branch is considered at every recursion, the algorithm end up with the length of the longest branch, which is the height of the tree. If the tree is made by just the root, the algorithm returns correctly 1.

Correctness of `RBTHEIGHT`

This algorithm just takes a tree T as input, find its root, apply to it `NODEHEIGHT`, and return the found tree height.

Complexity

As the algorithm visit all the nodes of the tree, performing constant time operations every time it is recursively called, its complexity is $\Theta(n)$ where n is the number of nodes in the tree.

c.

Algorithm 11: BHEIGHT: receives as inputs a tree, returns its height.

```
def BHEIGHT(x):
    g←0
    while x≠NIL do
        if x.color=BLACK then
            g←g+1
        end
        x←x.right
    end
    return g;
enddef
```

Algorithm 12: RBTBHEIGHT: receives as inputs a tree, returns its height.

```
def RBTBHEIGHT(T):
    x← T.root
    return BHEIGHT(x);
enddef
```

Correctness of `BHEIGHT`

Before the iteration: A node n of an RBT.
After the iteration: The algorithm go trough the rightmost branch of the tree, adding 1 to a counter every timec it encounters a BLACK node. Always going to the right child guarantees we will eventually reach a leave, and as all branches has the same BH, we are sure that our choice of branch is irrelevant for a correct output. If the tree is made by just the root, it is surely BLACK, and the algorithm returns correctly 1.

Correctness of `RBTBHEIGHT`

This algorithm just takes a tree T as input, find its root, applying to it `BHEIGHT`

Complexity

The time required by the algorithm depends on the number of RED nodes in the branch considered. In the worst case scenario, the one of a complete RBT, the complexity is $O(2\log_2(n+1))$. The best case scenario is the one in which the branch considered has only BLACK nodes, so $\Omega(\log_2(n+1))$. The overall complexity is then $\Theta(\log_2(n+1))$.

Exercise 4

a.

As different pairs can have the same *keys* a and b values, the use of BST and RBT is excluded. The next best thing is to adapt a *heap* to work with the condition imposed by the problem. If we defined the order $\preceq$ as the one required to lexicographically sort our pairs, we can store the pairs in a max-heap H_{max} , and at the moment of the sorting we can extract the root and fix the heap with heapify (as in heap sort). So, if we can access element a and b in a node x as $x.a$ and $x.b$, we can rewrite `HEAPIFY` as

Algorithm 13: HEAPIFY

```
def HEAPIFY(H, i):
    m ← i
    for j in [LEFT(i), RIGHT(i)] do
        if IS_VALID_NODE(H, j) and ( H[j].a > H[m].a or (H[j].a=H[m].a and
         H[j].b≥H[m].b) ) then
            m ← j
        end
    end
    if i != m then
        swap(H, i, m)
        HEAPIFY(H, m)
    end
enddef
```

For height h of a near complete binary heap we have $\lceil \frac{n}{2^{h+1}} \rceil$ nodes. What is the complexity of building the heap?
At each height h we have $\lceil \frac{n}{2^{h+1}} \rceil$ nodes. We want to fix the heap property by calling heapify on all the nodes of the tree. So the overall complexity, a.k.a. the complexity of

invoking Build_Heap (as defined in class) on n nodes

$$\begin{aligned} T_{BH}(n) &\leq \sum_{h=0}^{\lfloor \log_2 n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil \cdot O(h) \\ &\leq c \cdot n \cdot \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h} \\ &\leq c \cdot n \cdot \sum_{h=0}^{\infty} \frac{h}{2^h} \\ &\leq c \cdot n \cdot \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2 \cdot c \cdot n \end{aligned}$$

So

$$T_{BH}(n) \in O(n) \tag{4}$$

EXTRACT_MIN costs $O(\log i)$ per iteration, in which i is the number of elements in the heap (decreasing every time by 1). The total is

$$\begin{aligned} T_H(n) &= \Theta(n) + \sum_{i=2}^{n} O(\log i) \\ &\leq O(n) + O\Big(\sum_{i=2}^{n} \log n \Big) = O(n \log n) \end{aligned}$$

This is an upper-bound, for the worst case scenario. The actual number of iteration is different for every case.
The overall complexity of sorting in this way is then $O(n \log n)$.

b.

The conditions on a_i are the ones sufficient to apply counting sort. Let's consider an array A populated by all the pairs we are interested in. For a more in-depth explanation of counting sort and its complexity, go to section 5.6 of the `Algorithmic_Design_Notes` in this repository.
We could proceed in two steps:

- Apply counting sort on the a_i
- For every group of pairs sharing the same a, sort them based on their b_i using heap sort.

Grouping pairs with same a in a sorted array can be done in a time $\Theta(n)$ where n is the number of elements in the array. Calling for example *id* an attribute corresponding to

the name of the group a pair belongs to, we can write the pseudo-code:

Algorithm 14: GROUP: receives as inputs an array of pairs (a, b), it assigns the same *id* to pairs with the same a.

```
def GROUP(A):
    i←2
    k←1
    while i≠ |A| do
        if A[i].a=A[i-1].a then
            A[i].id←k
        end
        else
            k←k+1
            A[i].id←k
        end
    end
enddef
```

From which we could, for example, find the indexes ending points of every group to feed to Heap sort. Or, more directly

Algorithm 15: LEXSORT: receives as inputs an array of pairs (a, b), it performs on it a lexicologiacal sort.

```
def LEXSORT(A,k):
    COUNTING_SORT(A.a, B, k)
    i←2
    begin←0
    end←0
    while i≠ |A| do
        if B[i].a≠B[i-1].a then
            end←i-1
            HEAP_SORT(B[begin,end].b)
            begin←i
        end
    end
enddef
```

Where $Array[begin, end]$ is the sub-array of $Array$ starting at index *begin* and ending at index *end*, and $Array.a$ and $Array.b$ used in the functions means that $Array$ is sorted accordingly respectively to its components a or b (where obviously the corresponding algorithms was accordingly changed to do so). B is the sorted array.

Complexity

If for every array of length n we could choose accordingly a number k, than the best case scenario, the one in which all the a_i are different, is a viable option. The complexity of the

algorithm in the best case scenario would be the one of Counting Sort: $\Omega(n+k) = \Omega(n)$. In this context the algorithm would be an improvement with respect with the one proposed in point **a.**
However, if we keep k constant, and increase n, when we look at the asymptotic complexity, the best case scenario aforementioned is not a viable option no more, as progressively more pairs shares the same a. Its complexity then is the one of the innermost sorting algorithm (`HEAP_SORT` in this case), and the best and worst case scenario tends to match as $n \to \infty$. The complexity of the algorithm in our case is then $\Theta(n \log n)$, matching the one of the previous exercise, so with no gains in efficiency.

c.

Now the conditions necessary to apply `COUNTING_SORT` also helds for the b in the pairs. We can then apply the same process as the one in point **b.** to lexicographically sort the pairs:

- Apply counting sort on the a_i
- For every group of pairs sharing the same a, sort them based on their b_i using counting sort again.

Algorithm 16: LEXSORT: receives as inputs an array of pairs (a, b), it performs on it a lexicologiacal sort.

```
def LEXSORT(A, k1, k2):
    COUNTING_SORT(A.a, B, k1)
    i←2
    begin←0
    end←0
    while i≠ |A| do
        if B[i].a≠B[i-1].a then
            end←i-1
            COUNTING_SORT(B[begin,end].b, C, k2)
            begin←i
        end
    end
enddef
```

Where $Array[begin, end]$ is the sub-array of $Array$ starting at index $begin$ and ending at index end, and $Array.a$ and $Array.b$ used in the functions means that $Array$ is sorted accordingly respectively to its components a or b(where obviously the corresponding algorithms was accordingly changed to do so). C is the sorted array.

Complexity

As stated before, the complexity is the one of the innermost sorting algorithm (`COUNTING_SORT` in this case), and the best and worst case scenario match with complexity which is $\Theta(n+k)$, which, as $n \to \infty$ while keeping k constant, is $\Theta(n)$. In this case than there is a gain in efficiency.

Exercise 5

a.

Let's say we apply `Select` on an array A. Not containing duplicate values is not a necessary requirement for the algorithm `Select` to work. However, the complexity of the algorithm heavily depends on a balanced `PARTITION` of the array, and the strategy used to achieve it (i.e. using the median of the medians of all the chunks in which we divide the array) relies on this assumption. If we end up with two or more medians sharing the same value, we have to choose which one we should use as pivot.
A random pick could jeopardise the assumptions which we use to calculate the upper bounds on the number of elements smaller (or greater) than the median of the medians, and through that our calculation of the complexity (e.g. the case in which two chunks are all made by the the same number).
As it is, the algorithm could leads to sub-optimal partitions of the array , as all the elements with the same values are re-inserted in S, were the recursive call can lead the problem to came out again in subsequent iterations.
If we instead change the function `PARTITION` in such a way that all the values of the array are placed around the pivot, neither in S nor in G, they will surely be in the right position and will not cause any troubles in following iterations of the algorithm, keeping the increase in complexity to one instance (so without changing the asymptotic complexity).

b.

Algorithm 17: REP_PARTITION

```
def REP_PARTITION(A, i, j, p):
    swap(A, i, p)
    (p, i) ← (i, i+1)
    equal_elements←0
    while i≤j do
        if A[i] > A[p] then
            swap(A, i, j)
            j← j-1
        end
        if A[i] < A[p] then
            swap(A, i, p - equal_elements)
            i←i+1
            p←i
        end
        else
            i←i+1
            p←i
            equal_elements←equal_elements+1
        end
    end
    swap(A, p, j)
    return j-equal_elements, j
enddef
```

So the `Select` algorithm becomes:

Algorithm 18: SELECT

```
def SELECT(A, l=1, r=|A|, i):
    if r-l≤10 then // base case
        SORT(A, l, r)
        return i
    end
    j←SELECT_PIVOT(A, l, r)
    k, t←REP_PARTITION(A, l, r, j)
    if i=k then // dichotomic approach
        return k
    end
    if i<k then // search in S
        return SELECT(A, l, k-1, i)
    end
    // search in G
    return SELECT(A, t+1, r, i)
enddef
```