ADM-Homework 2
Theoretical Question

Clara Lecce (1796575)
Michele Luca Puzzo (1783133)
Gianmarco Ursini (1635956)

October 25, 2021

1 TQ 1

1.1 Question 1

As known, given a random variable X, the Quantile function $Q(\cdot)$ with support $\{p|p \in [0,1]\}$ is the function that computes:

$$Q(p) = s \mid \mathcal{P}(X <= s) = p \tag{1}$$

Denoting with A_i the i-th element of the vector A of length n and given $k \in [0, n]$, it is possible to see that our algorithm compute:

$$alg(A, k) = s \mid \#\{A_i <= s\} = k \tag{2}$$

It is then easily possible to perform some trasformations over our algorithm parameters in order to obtain the similarities with the quantile function (1), i.e.:

- A shrinkage over our algorithm support space (i.e. $k' = k/n$);
- A shrinkage over our cardinality measure (i.e. $\#\{A_i <= s\}' = \frac{\#\{A_i <= s\}}{n}$);

Substituting into (2) it becomes:

$$alg(A, k') = s \mid \frac{\#\{A_i <= s\}}{n} = k' \tag{3}$$

In a frequentist approach (said A_r a random sample of the vector A) we can equal $\frac{\#\{A_i <= s\}}{n} = \mathcal{P}(A_r <= s)$; In words, our algorithm is computing the value s so that the number of elements in the array A smaller or equal to s will be equal to k: we can so somehow define our algorithm a "quantile function over a non-normalized support".

1.2 Question 2

Let consider the worst case scenario, i.e. imagine that $k = n$ and that at each iteration the random sample s will always be equal to A_1: it basically means that the s satisfying the condition over k will be selected at the $n_{th} - 1$ iteration (when the vector A over which we are calling $alg()$ has lenght equal to 2), we can then assume an asymptotical complexity in the worst case scenario (removing costant therms) equal to $\mathcal{O}(n)$.

1.3 Question 3

In the best case scenario, the right s will always be picked up at the first iteration, regardless of n=len(A): the asymptotical complexity will then be equal to $\mathcal{O}(1)$.

2 TQ 2

2.1 Question 1

Let dive into the interpretation of the given recursive algorithm's complexity. It is clear that, given a particular n and $\forall l$, and expressing with $T(n)$ the time needed to complete the algorithm called with parameter n:

$$T(n) = T\left(\frac{n}{2}\right) \cdot 2 + \left(\frac{n}{2} + 1\right) \cdot 3 \tag{4}$$

Infact, calling `splitSwap(a,l,n)` we will have to solve two times `splitSwap(a,l,n/2)` plus execute 3 operations for each of the $(\frac{n}{2} + 1)$ iterations of the for loop into `swapList(a,l,n)`. Lets compute running times after (4):

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{2^2}\right) \cdot 2 + \left(\frac{n}{2^2} + 1\right) \cdot 3 \tag{5}$$

$$T(n) = T\left(\frac{n}{2^2}\right) \cdot 2^2 + \left(\frac{n}{2^2} + 1\right) \cdot 2 \cdot 3 + \left(\frac{n}{2} + 1\right) \cdot 3 \tag{6}$$

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

$$T\left(\frac{n}{2^2}\right) = T\left(\frac{n}{2^3}\right) \cdot 2 + \left(\frac{n}{2^3} + 1\right) \cdot 3 \tag{8}$$

$$T(n) = T\left(\frac{n}{2^3}\right) \cdot 2^3 + \left(\frac{n}{2} + 1\right) \cdot 3 \cdot 3 + 7 \tag{9}$$

$$T(n) = T\left(\frac{n}{2^k}\right) \cdot 2^k + \left(\frac{n}{2} + 1\right) \cdot k \cdot 3 + log_2(2^k) - 1 \tag{10}$$

Setting $2^k = n \Leftrightarrow k = log_2(n)$ we obtain:

$$T(n) = T(1) \cdot n + \left(\frac{n}{2} + 1\right) \cdot log_2(n) \cdot 3 + log_2(n) - 1 \simeq n \cdot log_2(n) \tag{11}$$

In (11) we have removed the dependency from factors, constant terms and considered only the term with the biggest growth rate w.r.t n. We can than say that the asymptotical complexity of the algorithm is $\mathcal{O}(n \cdot log_2(n))$.

2.2 Question 2

Given an array `a`, an index `l` and a number `n` (considering the scenario where both `len(a)` and `n` are power of 2 numbers), the algorithm output the array `a'` built as follows:

$$a'[i] = a[i] \qquad \forall i \in [0, 1, ..., l-1] \qquad \text{if} \qquad l \geq 1$$

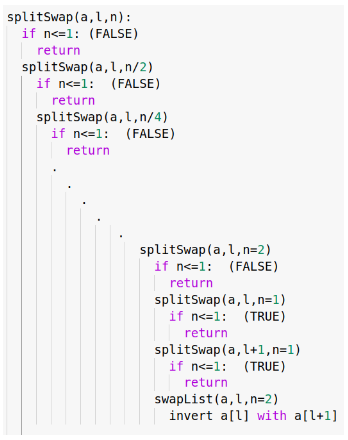
$$a'[l+i] = a[l+n-i]$$

In words, starting from an index `l` of the original array `a`, the algorithm is reversing the position of the first `n` elements of the array. Because of this of course it is required that `l+n` $\leq$ `len(a)`, otherwise the subroutine `swapList()` will raise an error because of the out-of-range index it loops on. Let describe the algorithm's mechanism. Looking at the code, we can assess how the only part of the code actually changing the position of the array's elements is the subroutine `swapList()`. Given a triplet `(a,l,n)`, once `splitSwap()` is called, it will recursively call himself with an `n` halfed call by call (i.e. $\texttt{n}^{(1)}$ =`n/2`, $\texttt{n}^{(2)}$ =$\texttt{n}^{(1)}$/2, $\texttt{n}^{(3)}$ =$\texttt{n}^{(2)}$/2 and so on). As we can see in the (Fig.1), after $log_2(n) - 1$ steps, the function `splitSwap(a,l,2)` will be called: in its execution both `splitSwap(a,l,1)` and `splitSwap(a,l+1,1)` will `return` (being `n`=1), finally allowing the execution of `swaplist(a,l,2)` (that we will call `final-node-subroutine` $\forall l$) that will exchange the position of the array's elements `a[l]` with `a[l+1]`. Being `splitSwap(a,l,2)` completed, `splitSwap(a,l+2,2)` will be called. Similary, at the end of the execution its `final-node-subroutine` will exchange the position of the array's elements `a[l+2]` with `a[l+3]`. Basically the `final-node-subroutines` consider the array (starting from the element $a[l]$) as a sequence of $\frac{n}{2}$ couples of elements and in each couple they exchange the

1st element with the 2nd one.
Recalling that `splitSwap(a,l,2)` and `splitSwap(a,l+2,2)` where called in `splitSwap(a,l,4)`, `swapList(a,l,4)` (that we will call `semi-final-node-subroutine`) will finally be executed, exchanging the position of the array's elements `a[l]` with `a[l+2]` and `a[l+1]` with `a[l+3]`. So the role of `semi-final-node-subroutines` is to consider the array (starting from the element $a[l]$) as a sequence of $\frac{n}{4}$ couples of couples and to exchange the position of the 1st element of the 1st couple with the 1st element of the 2nd couple, and the 2nd element of the 1st couple with the 2nd element of the 2nd couple. Basically, after the execution of all the `final-node-subroutines` and of the `semi-final-node-subroutines` the position of the 1st group of 4 elements of the original array will be reversed, the same for the 2nd group of 4 elements and so on. We can so climb our recursive function tree from the `final-node-subroutines` up to the top `first-final-node-subroutine` i.e. `swapList(a,l,n)`. We can see the effect of each kind of `subroutine` level over a test array in two examples at (Fig.2,3) recalling that the output of the `first-final-node-subroutine` will be equal to the algorithm's output.
Having assessed that the algorithm complexity is $\simeq O(n \cdot log_2(n))$, it is possible to confirm that the algorithm it's not optimal: infact it is easily possible to write some pseudo-code with a lower complexity than the given algorithm:

```
def reverse(a,l,n):
   reversedarray=a
   for i in range(n):
      reversedarray[i+l]=a[l+n-i]
   return reversedarray
```

We can easily see that the `reverse()` algorithm complexity has now become (removing costant therms and factors) $O(n)$, proving that the `splitSwap()` algorithm was not optimal.



```
splitSwap(a,l,n):
  if n<=1: (FALSE)
    return
  splitSwap(a,l,n/2)
    if n<=1:  (FALSE)
      return
    splitSwap(a,l,n/4)
      if n<=1:  (FALSE)
        return
      .
        .
          .
            .
               .
                 splitSwap(a,l,n=2)
                   if n<=1:  (FALSE)
                     return
                   splitSwap(a,l,n=1)
                     if n<=1:  (TRUE)
                       return
                   splitSwap(a,l+1,n=1)
                     if n<=1:  (TRUE)
                       return
                   swapList(a,l,n=2)
                     invert a[l] with a[l+1]
```

Figure 1: Reaching the first `final-node-subroutine`

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]   <-----   Original test array a, len(a)=n=16, l=0
[1, 0, 3, 2, 5, 4, 7, 6, 9, 8, 11, 10, 13, 12, 15, 14]   <-----   Effect of final-node-subroutines swapList(a,0,2)
[3, 2, 1, 0, 7, 6, 5, 4, 11, 10, 9, 8, 15, 14, 13, 12]   <-----   Effect of semi-final-node-subroutines swapList(a,0,4)
[7, 6, 5, 4, 3, 2, 1, 0, 15, 14, 13, 12, 11, 10, 9, 8]   <-----   Effect of semi-semi-final-node-subroutines swapList(a,0,8)
[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]   <-----   Effect of first-final-node-subroutines swapList(a,0,16)
```

Figure 2: Test over a with len(a)=n=16, l=0

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]   <-----   Original test array, len(a)=16, n=8, l=7
[0, 1, 2, 3, 4, 5, 6, 7, 9, 8, 11, 10, 13, 12, 15, 14]   <-----   Effect of final-node-subroutines swapList(a,7,2)
[0, 1, 2, 3, 4, 5, 6, 7, 11, 10, 9, 8, 15, 14, 13, 12]   <-----   Effect of semi-final-node-subroutines swapList(a,7,4)
[0, 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11, 10, 9, 8]   <-----   Effect of first-final-node-subroutines swapList(a,7,8)
```

Figure 3: Test over a with len(a)=16, n=8, l=7

3 Knapsack

In this theoretical question we have to face with a NP-complete problem: the Knapsack one. To solve it generally we have to use heuristic solutions but in some cases they fail to provide the optimal solution.

- The first heuristic solution is a greedy algorithm in which we order the object in increasing order of weight and then visit them sequentially, adding them to the solution as long as the budget is not exceeded. This algorithm does not provide the optimal solution in every situation indeed in my counterexample this greedy algorithm fails: we fix the budget: $\mathbf{W} = 10$ and we have three object.

i	$\mathbf{w}_i$	$\mathbf{v}_i$
1	4	3
2	6	5
3	10	9

  We have to visit the object sequentially so we are going to pick the first two objects, but we cannot pick the third one because we will exceed the budget. This choice is not optimal because it would be better pick only the third object because its values (9) is greater of the sum of the first two (8).

- In the second heuristic solution we have to order the objects in decreasing order of values, and then visit them sequentially, adding them to the solution if the budget is not exceeded. This algorithm does not provide the optimal solution in each situation indeed in my counterexample this greedy algorithm fails: I have decided to choose the same budget $\mathbf{W} = 10$ and the same number of object of the last counterexample.

i	$\mathbf{w}_i$	$\mathbf{v}_i$
1	9	9
2	7	7
3	3	3

  We have to visit the objects sequentially so we are going to pick the first object, but we cannot pick the last two because we will exceed the budget. This choice is not optimal because it would be better pick the second and the third objects because the sum of their values (10) is greater of the first object value (9).

- In the third heuristic solution we have to order them in decreasing relative value ($v_1/\ w_i$), and then visit them sequentially, adding them to the solution if the budget is not exceeded This algorithm does not provide the optimal solution in each situation indeed in my counterexample this greedy algorithm fails: I have decided to choose the same budget $\mathbf{W} = 10$ and the same number of object of the two last counterexamples.

i	$\mathbf{w}_i$	$\mathbf{v}_i$
1	9	10
2	7	7
3	3	3

  We have to visit the objects sequentially so we are going to pick the first object whose relative value is 1.11 while the one of the other objects is 1. We cannot pick the last two because we will exceed the budget. This choice is not optimal because it would be better pick the second and the third objects because the sum of their values (10) is greater of the first object value (9).