Assignment 1
Algorithms for Big Data

Lavinia Pulcinella
Student ID: i6233926

February 2020

1 Exercise 1

```
i := 0; j := 0; i := 0; j := 0
sum:=0; maxsum := sum;
while j ≤ n − 1 do
    if sum + a[j] > 0 then
        sum = sum + a[j];
        j = j+1;
    else
        sum = 0
        j = j+1 ; i=j;
    end
    if sum > maxsum then
        maxum=sum
        i* = i ; j* = j-1
    else
    end
end
return i*, j*
```

Algorithm 1: Find Optimum Subarray

We want to prove that our algorithm correctly returns a maximum subarray whenever such solution is positive.
We can prove that the algorithm works by induction.
As a base case we consider when n=1 i.e. when the array has just one element. It is clear that in this case the algorithm correctly returns the maximum.

We now suppose that the algorithm returns the maximum positive sum for arrays of any length i.e. n $>$ 1.
The algorithm is iterated for every j within the length of the array (i.e. n). As j increases the sum is updated with the j-th element value if the sum is positive the value is updated and j is updated. Otherwise, both i-th and j-th indices are updated. If the new sum is bigger than the maximum sum computed up to the j-th point - the optimal indices i^* and j^* are updated.
Thus, the maximum array is computed.

Morover the if statements within the while loop have all constant running time i.e. $O(1)$. The while loop however is iterated n-1 times for true contition and one more for the false condition, thus for a total of n times. Hence, the algorithm running time is $O(n)$

2 Graded exercise 2

The BigJump problem asks, for a given array a of n elements, for a longest jump in the array.
A jump in an array is a pair of indices $(i; j), i \leq j, i; j \in \{0, ..., n-1\}$**. The length of a jump** $(i; j)$ **is** $a[j] - a[i]$**.**

While such a problem can be easily dealt with by using brute force we would obtain a quadratic running time. We can do better by using the Divide&Conquer technique. This design implies the input array a to be recursively divided into two or more sub-problems which are easier to solve. The solutions are then combined in order to provide the solution to the problem.

In this case, splitting the array in two parts namely *LEFT* and *RIGHT* we end up having three possible solutions:

1. i & $j \in$ *LEFT*
2. i & $j \in$ *RIGHT*
3. $i \in$ *LEFT* & $j \in$ *RIGHT*

We can solve this problems recursively. Specifically, solution (3) can be dealt with by searching for the minimum value on the *LEFT* half (i.e. min_{LR}) and the maximum one on the *RIGHT* half (i.e. max_{LR}). This would guarantee that $j > i$.

The recursive process implies that min_{LR} and max_{LR} are updated each time the array is merged back together.

function big _ jump(a[i..j]) : returns longjump =max(a[j] - a[i]);
if i=j **then**
  longjump =0;
else
  M = $\lceil n/2 \rceil$
  $\text{jump}_L = big\_jump(a[0 \,..\, M])$
  $\text{jump}_R = big\_jump(a[M+1 \;..\; n-1])$
  $\text{min}_{LR} = min(a[0 \,..\, M])$
  $\text{max}_{LR} = max(a[M+1 \;..\; n-1])$
  $\text{jump}_{LR} = max_L R - min_L R$
  longjump = max($\text{jump}_L, jump_R, jump_{LR}$)
end
return *longjump*

Algorithm 2: BIGJUMP Divide&Conquer

As already said, Divide&Conquer technique implies our problem to be divided into two or more sub-problems for a total of *logn* levels. In the algorithm this is represented by the recursion of the big _ jump function. Moreover the min and max functions find min_LR and max_LR respectively and they each require n/2 computations.Thus, the algorithm has a running time of $O(nlogn)$.

3 Exercise 3

Solving the BIGJUMP problem via Dynamic Programming allows to simplify it by dealing with more sub-problems which are easier to solve.
We can set the first value of the array as the minimum one. Then we iterate in order to check if the i+1 -th value is smaller than the one initialized, if its not we compute the difference between i-th value and and the minimum one. We then compare this final result to the longest jump computed so far (initialized at 0) and we updated if the result is bigger thant the last one.

```
minvalue = a[0] ;
longjump = 0 ;
for i ← 0 to n − 1 do
    if a[i] < minvalue then
        minvalue = a[i] ;
    end
    else if a[i] - minvalue > longjump then
        longjump = a[i] - minvalue ;
    end
end
return longjump
```

Algorithm 3: BIGJUMP Dynamic Programming

Since the if statements within the for loop have a linear running time and the for loop works along the n elements of the array, the total running time of the algorithm is $O(n)$.