DMA assignment 2

Mikkel *Luc* Carlsen

15. september 2020

Note to reader! Each task is given its own page in this assignment.

Task A

In order to figure out how many inversions there is in the Array A, then it will be necessary to look at each value n_i and compare it to all other values n_j such that $n_i > n_j$ and $i < j$. I have done so in the following table:

	$6_{i=0}$	$2_{i=1}$	$1_{i=2}$	$6_{i=3}$	$4_{i=4}$	$8_{i=5}$	$10_{i=6}$	i
$6_{j=0}$	false	false	false	false	false	false	false	
$2_{j=1}$	true	false	false	false	false	false	false	
$1_{j=2}$	true	true	false	false	false	false	false	
$6_{j=3}$	false	false	false	false	false	false	false	
$4_{j=4}$	true	false	false	true	false	false	false	
$8_{j=5}$	false	false	false	false	false	false	false	
$10_{j=6}$	false	false	false	false	false	false	false	
j								
total	3	1	0	1	0	0	0	

Tabel 1: This table is handmade and has the purpose of showing which values of $n_i > n_j$ and $i < j$. Each cell in the table has the value "true" if the demands for an inversion is met, otherwise it has the value "false". The bottom row of the table has the title "total" and shows how many inversions there is regarding the current i

Through table 1 it can be concluded that there is 5 inversions, i.e. if the case were to use a sorting algorithm on the array, then it would need to perfrom 5 inversions/5 steps in order to have sortet the array.

Task B

In order to figure out the highest amount of inversions the array A can possible have, then it is necessary to create the demand that A is always reverse sorted, otherwise each number n wont be able to have its maximum amount of inversions. Now assuming that the array A is reverse sorted, then the amount of inversions for each n can be described with the following expression:

$$\sum_{k=0}^{n-1} n_k \tag{1}$$

Example of the expression. If the reverse sorted array is [5,4,3,2,1,0] then we have n=6 values in the array A... therefor our expression for n=6 would look like:

$$\sum_{k=0}^{n=6-1} n_k = 0 + 1 + 2 + 3 + 4 + 5 \tag{2}$$

This might look confusing at first sight due to the fact that this summation consist of the same numbers as the array but reversed again. This is caused by the way the summation notation works and the fact that the array has to be reversed sorted in order for $i < j$ and $A[i] > A[j]$ as many times possible for each n. I.e. in the array [5,4,3,2,1,0] then 5 has 5 inversions, because 5 is the first number and is bigger than all the other numbers. If the array weren't reverse sorted but just sorted normally, then 5 would still be bigger than all the other numbers but it would no longer fulfill the demand that $i < j$ and therefor have 0 inversions, same would follow for all the other numbers(in an exaggerated meaning due to 4 not being bigger than 5 and 3 not being bigger than 4 and 5 etc.).

Task C

Pseudocode that would count the amount of inversions could look like one of the two following:

```
SIMPLE_INVERSION_COUNTER(I,n)
1    INV = 0
2    for i = 0 to n-1
3        for j = 0 to n-1
4            if i<j and I[i] > I[j]
5            INV = INV+1
6    return INV
```

```
ADVANCED_INVERSION COUNTER(I,n)
1    INV = 0
2    for i = 0 to n-2
3        for j = i+1 to n-1
4            if I[i] > I[j]
5            INV++
6    return INV
```

What the SIMPLE_INVERSION_COUNTER does is that in line 1 it starts with setting the variable INV to 0, which is the variable it will use to count the amount of inversions. In lines 2 and 3 it creates a for-loop and a for-loop within the previous one. Both for-loops run from 0 to n-1 but they use a different variable. The first for-loop in line 2 uses the variable i and the second for-loop in line 3 uses the variable j. Now what this does is that in line 4, it checks if the demands for an inversion is met by the values of i and j which is set in the for-loops. Now in line 5, if the demands are met then it adds 1 to the variable INV. In the end the algorithm will in line 6 return the value of the variable INV and thereby tell how many inversions were made.

What the ADVANCED_INVERSION_COUNTER does is basically the same as SIMPLE_INVERSION_COUNTER, but faster due to the fact that it just leaves out unecessary values. For example in line 2 the for-loop goes from 0 to n-2, that is because the last number will never fullfill the demand that $i<j$. Moreover the ADVANCED algorithm also sets the second for-loop in line 3 to run from $i+1$ to n-1 instead. Doing this allows the if statement to simply check for if $I[i] > I[j]$ and is easier readable. Also the algorithm doesn't waste time checking unecessary values that would not fullfill the demand $i<j$. And as a final touch the ADVANCED algorithm uses the $++$ statement instead of saying the variable itself$+1$, which is shows better understanding of writing algorithms in Pseudocode and again is easier to read.

Task D

The algorithm that will be analysed is the ADVANCED_INVERSION_COUNTER. The algorithm will be below in order to avoid the necessarity of scrolling up and down between this page and the page before. Moreover the running time of each line will be written to the right of the line, as showed in the following:

1 INV = 0	c_1
2 for i = 0 to n-2	$(n-1)$
3 for j = i+1 to n-1	$\sum_{k=0}^{n-1} n_k$
4 if I[i]>I[j]	c_2
5 INV++	c_3
6 return INV	c_4

From the above written running times of each line in the ADVANCED algorithm it can be concluded that the total running time of the algorithm is:

$$c_1 + (n-1) + \sum_{k=0}^{n-1} cn_k + c_2 + c_3 + c_4 \tag{3}$$

which is the same as:

$$c_1 + c_2 + c_3 + c_4 + (n-1) + \sum_{k=0}^{n-1} n_k \tag{4}$$

This can finally be rewritten to:

$$\sum_{k=0}^{n-1} n_k + (n-1) + c_7 \tag{5}$$

In equation 5 $c_7 = c_1 + c_2 + c_3 + c_4$.
The running time would then be written as $\Theta(n^2)$ due to the fact that $\sum_{k=1}^{n} k = \Theta(n^2)$[1] which is the same as $\sum_{k=0}^{n-1} n_k$, because the lower bound is 1 less and so is the upperbound, therefor the difference is 0 and the variable k has just been replaced by n_k.

[1] This notation is from the book CLRS at page 1146(digital edition), statement A.2