Maximilien Malderle
ID: 26562906
Assignment 1 352

Nr. 1

a) Pseudo Code:

```
Algorithm: Swap until middle then every calculate second (Swap and Calc)
Input: An array of Integers size n>= 1 called a
Output: Modified Array a
middle <— 0
i <— 0
j <— 1
temp <— 0

if a[] length % 2 == 0
        then middle <— (a[] length / 2 ) -1
else
        then middle <— (a[] length / 2 )

while i <= middle and j <= middle
        Swap a[i] and a[j] using temp
        increment i + j by 2

i <— middle +1
j <— i +1

while j < a[] length
        a[j] = a[i] + a[j]
        increment i + j by 2
```

b)

The time complexity is O(1) for the if test, O(1) for the allocation of middle, O(n/4) for the first while loop (half the array always using 2 spots per iteration) and O(n/4) for the second while loop (same reason). When adding all these together we can estimate a O(n/2). This means that we only iterate through half the array, because each iteration includes 2 items. This is the same as saying O(n), which is linear.

c) The space complexity is O(n) for the array and 0(4) for the variables middle, i, j and temp. Yet because the array is passed to the function the function does not need to allocate the memory for this.
This means that the space complexity of the algorithm is O(4) which is the same as O(1) which means constant.

Nr 2.

a)

O(n^7) is a valid but not the best time complexity for this process

2n^5 log n < 2(n^7 log n) with c=2 and n>1

But:
with n = 2

2 * 2^5 —> 2^6

and with n = 3

2 * 3^5 < 3^6

so the highest the algorithm can go to is n^6 which would be more accurate than n^7

b)

This is true because if c1=1 for n<=2

1*(2^6) < 10^8 * n^2 + 5*n^4 + 7000 * n^3 + n

and if c2= 10^8 + 5 +7000 for n<=2 then

10^8 * n^2 + 5*n^4 + 7000 * n^3 + n < (10^8 + 5 + 7000)*(n^6)

c)

This is false because there is no constant c that makes n! always greater than n^n.

n^n with n=5 —> 5^5 = 3125
n! with n=5 —> 5! = 640

3125 / 640 = 4.88

so n^n < 5 *5! for n>1

but

6^6 !< 5* 6! —> 46 656 !< 9720

because n^n is exponential and n! is not that means O(n!) is not valid for n^n

d)

for c1=0.0000001 and n>= 2

0.0000001* n^3 < 0.01 n^3 + 0.0000001 n^7

but for c2 = (0.01 + 0.0000001) with n>=2

it is impossible for this to be greater than 0.01 n^3 + 0.0000001 n^7

because of the term n^7 which will always produce higher values than n^3

This big theta statement is false

e)

This is true because

n^2 + 0.0000001 n^5 > (1 + 0.0000001) n^3 for n<=2

f)

This is true because

n! with n = 4 is 4*3*2*1 —> 2*2*2*3*1
2^n with n = 4 is 2^4 —> 2*2*2*2

so for n<4 we need a c < 2

2* 3! = 2*3*2*1
2^3 = 2*2*2

2*2! = 2*2*1
2^2 = 2*2

so n! > 1/3* 2^n

Nr. 3

a) O() —> First while loop worst case —> n-2
—> Second while loop worst case —> n-2
—> This means that the big o —> is roughly O(2n) —> O(n)
—> The last function call recalls the original function resulting in O(n^2)

Omega() —> is the same as the big O only that the last function is only called once so
—> OMEGA(n).

b)

1 run: 1st while loop —> beg: A{4, 10, 5, 1, 3}
—> j = 0 —> no movement
—> j = 1 —> swap 10 and 5 —> A{4, 5, 10, 1, 3}
—> j = 2 —> swap 10 and 1 —> A{4, 5, 1, 10, 3}
—> j = 3 —> no movement

2nd while loop —> beg: A{4, 5, 1, 10, 3}
—> j = n-1 —> swap 10 and 3 —> A{4, 5, 1, 3, 10}
—> j = n-2 —> no movement
—> j = n-3 —> swap 1 and 5 —> A{4, 1, 5, 3, 10}
—> j = n-4 —> no movement

2nd run: 1st while loop —> beg: A{4, 1, 5, 10, 3}
—> j = 0 —> swap 4 and 1 —> A{1, 4, 5, 3, 10}
—> j = 1 —> no movement
—> j = 2 —> swap 5 and 3 —> A{1, 4, 3, 5, 10}
—> j = 3 —> no movement

2nd while loop —> beg: A{1, 4, 3, 5, 10}
—> j = n-1 —> no movement
—> j = n-2 —> no movement
—> j = n-3 —> swap 4 and 3 —> A{1, 3, 4, 5, 10}
—> j = n-4 —> no movement

Terminated

c) This is a sorting algorithm, it sorts the contents of the array from smallest to largest. For any array size n, the algorithm will place the smallest value at position [0] and then increment the next smallest number until the largest number is stored at the end of the array.

d) Yes the run time can be easily reduced by returning the array without the extremities, because after one run through (both loops) the two extreme points of the array contain the smallest and largest numbers. This means the next iteration can use array n-2 making the next run through less long.

e) This is a linear algorithm, because there is only one recursive call made in the algorithm, but it is not tail recursive, because the recursive call is not made at the end, if it were to be tail recursive the call would need to be made at the return statement.