Assignment_3

February 23, 2024

1 Assignment 3

Peder Ørmen Bukaasen, Bård Tollef Pedersen and Eivind Lid Trøen

1.1 Exercise 1

```
def f1(n):
    result = [1 for _ in range(10)]  #  constant time c*10
    return result
```

Here the time complexety is c*10, and can be written as O(1)

```
def f2(n):
    total = 0
    for i in range(n): # c1*n
        total += i # constant time c2
    return total
```

Here the time complexety is c1*n+c2, and can be written as O(n)

```
def f3(n):
    matrix = [[0 for _ in range(n)] for _ in range(n)]  # c1*n * c2*n
    return matrix
```

Here the time complexety is c1 * n * c2 * n, and can be written as c3 * n * n wich is equal to O(n^2)

```
def f4(n):
    if n <= 1:  # constant time c1
        return n
    count = 0
    for _ in range(2**n): # c2 * 2^n
        for _ in range(2**n): # c3 * 2^n
            count += 1 # constant time c4
    return count
```

Here the time complexety is c1 + c2 * 2^n * c3 * 2^n + c4, and can be written as c5 + c6 * 2^n * 2^n, can be written ad c5 + c6 * 4^n wich is equal to O(4^n)

1.2 Exercise 2

Use mathematical induction to show that when is an exact power of 2, the solution of the recurrence

T(n) = {
1 if n = 1
8T(n/2) + n2 if n = 2k for k >= 1
} is T(n) = n2 * (2n - 1)

First prove that it holds for n=21

T(21) = (21)2 * (2 * 21 -1) = 22 * 22 - 22 = 16 - 4 = 12
T(21) = 8T(21 /2) + 21 2 = 8 * T(1) + 4 = 8 * 1 + 4 = 12

Then asume it is true for n=2k:
T(2k) = (2k)2 (2(2k) - 1)

And want to prove that T(2k+1) = (2k+1)2 * (2 * 2k+1 - 1).

T(2k+1) = 8T(2k+1 /2) + (2k+1)2
T(2k+1) = 8T(2k) + (2k+1)2
T(2k+1) = 8((2k)2 * (2(2k) - 1)) + (2k+1)2
T(2k+1) = 8($(2k)^2$ * (2k+1 - 1)) + (2k+1)2
T(2k+1) = 8(22k * (2k+1 - 1)) + (2k+1)2
T(2k+1) = 8(22k * 2k+1 - 22k) + (2k+1)2
T(2k+1) = 23 * 22k * 2k+1 - 23 * 22k + 22k+2
T(2k+1) = 22k+3 * 2k+1 - 22k+2
T(2k+1) = 23k+4 - 22k+2
T(2k+1) = 22k+2 * 2k+2 - 22k+2
T(2k+1) = 22k+2 * (2k+2 - 1)
T(2k+1) = (2k+1)2 (2 * 2k+1 - 1)

Q.E.D

1.3 Exercise 3

Master theorem states that given a recurrence relation of the form:

T(n) = aT(n / b) + Θ (nd)

where:
“a” is the number of subproblems in the recursion,
“b” is the factor by which the input size is reduced in each recursive call,
“Θ (nd)” is the cost of the work done outside the recursive calls.

The Master Theorem provides asymptotic bounds for the solution “T(n)” based on the properties of “Θ(n)”:

If d > log_b(a):
Then T(n) = Θ (nd).

If d = log_b(a):
Then T(n) = Θ (nd * log(n)).

If d < log_b(a):
Then T(n) = Θ (nlog_b(a)).

a) T(n) = 2T(n/4) + 1

Her a=2, b=4 and d=0
d = 0
log_b(a) = log_4(2) = 0.5
d < log_b(a)
T(n) = Θ (nlog_b(a))
T(n) = Θ (nlog_4(2))
T(n) = Θ ($\sqrt{n}$)

b) T(n) = 2T(n/4) + sqrt(n)

Her a=2, b=4 and d=0.5
d = log_b(a)
T(n) = Θ ($\sqrt{n}$ * log(n))

c) T(n) = 2T(n/4) + n

Her a=2, b=4 and d=1
d > log_b(a)
T(n) = Θ(n1)
T(n) = Θ(n)

d) T(n) = 2T(n/4) + n2

Her a=2, b=4 and d=2
d > log_b(a)
T(n) = Θ(n2)