### Practical Session #3 - Recursions

<table>
<thead>
<tr>
<th>Substitution method</th>
<th>Guess the form of the solution and prove it by induction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration Method</td>
<td>Convert the recurrence into a summation and solve it</td>
</tr>
<tr>
<td>Master Method</td>
<td>Tightly bound a recurrence of the form: ( T(n) = aT(n/b) + f(n) ), where ( a \geq 1, b &gt; 1 ), and ( f(n) ) is a asymptotically positive:</td>
</tr>
</tbody>
</table>
|                     | 1. if \( f(n) = O(n^{\log_b a - \epsilon}) \) for some \( \epsilon > 0 \), then \( T(n) = \Theta(n^{\log_b a}) \).  
                     | 2. if \( f(n) = \Theta(n^{\log_b a}) \), then \( T(n) = \Theta(n^{\log_b a \log n}) \).  
                     | 3. if \( f(n) = \Omega(n^{\log_b a + \epsilon}) \) for some \( \epsilon > 0 \), and there exist \( c < 1 \) for which \( a^* f(n/b) \leq c^* f(n) \) for all large enough \( n \) values, then \( T(n) = \Theta(f(n)) \).  
                     | Note: Generalized Case 2: if \( f(n) = \Theta(n^{log_b a log k n}) \), for some \( k \geq 0 \), then \( T(n) = \Theta(n^{log_b a log(k+1)n}) \). |

### Question 1: substitution method example

\[
T(1) = 1 \\
T(n) = T(\frac{n}{2}) + \sqrt{n}
\]

Prove that \( T(n) = O(\sqrt{n}) \) using the substitution method.

**Solution:**

\[
T(n) = T(\frac{n}{2}) + \sqrt{n} = T(\frac{n}{4}) + \sqrt{\frac{n}{2}} + \sqrt{n} = \ldots = T(\frac{n}{2^i}) + \sqrt{\frac{n}{2^{i-1}}} + \ldots + \sqrt{n}
\]

(Note: The 2\textsuperscript{nd} and 3\textsuperscript{rd} equations are not part of the substitution method, they are there just so we get the hang of the iteration)

**Guess:** \( T(n) = O(\sqrt{n}) \), meaning \( T(n) \leq c \sqrt{n} \)

**Induction base:** \( n = 1, c = 4 \)

\( T(1) = 1 \leq c \sqrt{1} \), \( c > 0 \)

**Induction step:**

\[
T(n) = T(\frac{n}{2}) + \sqrt{n} \leq c \sqrt{\frac{n}{2}} + \sqrt{n} = \sqrt{n} \left( \frac{c}{\sqrt{2}} + 1 \right) \leq c \sqrt{n}
\]

\((c/\sqrt{2} + 1) \leq c \) true for \( c = 4 \)
Question 2: iteration method example

Consider the following recurrence:

\[ T(a) = \Theta(1) \]
\[ T(n) = T(n-a) + T(a) + n \]

Find \( T(n) \) using the iteration method.

Solution:

\[ T(n) = T(n-a) + T(a) + n = [T(n-2a) + T(a) + (n-a)] + T(a) + n \]
\[ = [T(n-3a) + T(a) + (n-2a)] + 2T(a) + 2n - a \]
\[ = T(n-3a) + 3T(a) + 3n - 2a - a \]
\[ = [T(n-4a) + T(a) + (n-3a)] + 3T(a) + 3n - 2a - a \]
\[ = T(n-4a) + 4T(a) + 4n - 3a - 2a - a = \ldots \]
\[ = T(n-ia) + i * T(a) + i * n - \sum_{k=1}^{i-1} (k * a) \]
\[ = T(n-i*a) + i * T(a) + i * n - a * \frac{i * (i-1)}{2} \]

After \( \left( \frac{n}{a} - 1 \right) \) steps, the iterations will stop, because we have reached \( T(a) \).

Assign \( i = \frac{n}{a} - 1 \):

\[ T(n) = T \left( n - \left( \frac{n}{a} - 1 \right) a \right) + \left( \frac{n}{a} - 1 \right) T(a) + \left( \frac{n}{a} - 1 \right) n - a \left( \frac{1}{2} \left( \frac{n}{a} - 1 \right) \left( \frac{n}{a} - 2 \right) \right) \]
\[ = \frac{n}{a} T(a) - T(a) + \frac{n^2}{a} - n - \frac{a}{2} \left( \frac{n^2}{a^2} - \frac{3n}{a} + 2 \right) \]
\[ = \frac{n}{a} T(a) + \frac{n^2}{a} - n - \frac{n^2}{2a} + \frac{3n}{2} - a = \frac{n}{a} T(a) + \frac{n^2}{2a} + \frac{n}{2} - a \]
\[ = \Theta(n) \Theta(1) + \Theta(n^2) + \Theta(n) = \Theta(n^2) \]

\[ T(n) = \Theta(n^2) \]

Question 3: master method example
Use the Master method to find \( T(n) = \Theta(\cdot) \) in each of the following cases:

a. \( T(n) = 7T(n/2) + n^2 \)

b. \( T(n) = 4T(n/2) + n^2 \)

c. \( T(n) = 2T(n/3) + n^3 \)

Solution:

a. \( T(n) = 7T(n/2) + n^2 \)

\[ a = 7, \quad b = 2, \quad f(n) = n^2 \]
\[ n^{\log_2 7} = n^{2.803\ldots} \]
\[ n^2 = O(n^{\log_2 7 - \varepsilon}) \text{ since } n^2 \leq cn^{2.803\ldots - \varepsilon}, \quad c > 0 \]
\[ T(n) = \Theta(n^{\log_2 7}) \]

b. \( T(n) = 4T(n/2) + n^2 \)

\[ a = 4, \quad b = 2, \quad f(n) = n^2 \]
\[ n^2 = \Theta(n^{\log_2 4}) = \Theta(n^2) \]
\[ T(n) = \Theta(n^{\log_2 4 \log n}) = \Theta(n^2 \log n) \]

c. \( T(n) = 2T(n/3) + n^3 \)

\[ a = 2, \quad b = 3, \quad f(n) = n^3 \]
\[ n^{\log_3 2} = n^{0.63\ldots} \]
\[ n^3 = \Omega(n^{\log_3 2 + \varepsilon}) \text{ since } n^3 \geq cn^{0.63\ldots + \varepsilon}, \quad c > 0 \]
\[ \text{and} \quad 2f(n/3) = 2n^3/27 \leq cn^3 \text{ for } 2/27 < c < 1 \]
\[ T(n) = \Theta(n^3) \]

Question 4: design and analysis of algorithms for Fibonacci series

Fibonacci series is defined as follows:
\[ f(0) = 0 \]
\[ f(1) = 1 \]
\[ f(n) = f(n-1) + f(n-2) \]

Find an iterative algorithm and a recursive one for computing element number \( n \) in
Fibonacci series, Fibonacci(n).
Analyze the running-time of each algorithm.

Solution:

a. Recursive Computation:

```c
recFib(n) {
    if (n ≤ 1)
        return n
    else
        return recFib(n-1) + recFib(n-2)
}
```

T(n) = T(n-1) + T(n-2)+1
2T(n-2) +1 ≤ T(n-1) + T(n-2)+1 ≤ 2T(n-1) +1

T(n) ≥ 2T(n-2) +1 ≥ 2(2T(n-4) +1) +1 ≥ 2(2(2T(n-6) +1) +1) +1 ≥...

≥ 2^kT(n-2k)+\sum_{i=0}^{k-1}2^i.

The series end when k = \frac{n-1}{2}.

\rightarrow T(n) ≥ 2^{n-1} + 2^{n-2} -1 = \Omega(2^{n/2})

Same way:

T(n) ≤ 2T(n-1) +1 ≤ 2(2T(n-2) +1) +1 ≤ 2(2(2T(n-3) +1) +1) +1 ≤...

≤ 2^kT(n-k)+\sum_{i=0}^{k-1}2^i

The series end when k=n-1

\rightarrow T(n) ≤ 2^{n-1} + 2^{n-1} -1= O(2^n)

T(n) = O(2^n)
T(n) = \Omega(2^{n/2})
b. **Iterative Computation:**

Iterative computation:

```java
IterFib (n) {
    f[0] = 0;
    f[1] = 1;
    for ( i=2 ; i ≤ n ; i++)
        f[i] = f[i-1] + f[i-2];
}
```

\[ T(n) = O(n) \]

**Question 5: analysis of algorithm for the Hanoi Towers problem**

**Hanoi towers problem:**

n disks are stacked on pole A. We should move them to pole B using pole C, keeping the following constraints:

- We can move a single disk at a time.
- We can move only disks that are placed on the top of their pole.
- A disk may be placed only on top of a larger disk, or on an empty pole.

Analyze the given solution for the Hanoi towers problem; how many moves are needed to complete the task?
Solution:

```java
static void hanoi(int n, char a, char b, char c){
    if(n == 1)
        System.out.println("Move disk from " + a + " to " + b +"\n");
    else{
        hanoi(n-1,a,c,b);
        System.out.println("Move disk from " + a + " to " + b +"\n");
        hanoi(n-1,c,b,a);
    }
}
```

T(n) = the number of moves needed in order to move n disks from tower A to tower B.

- T(n-1)
  Number of moves required in order to move n-1 disks from tower A to tower C
- 1
  One move is needed in order to put the largest disk in tower B
- T(n-1)
  Number of moves required in order to move n-1 disks from tower C to tower B

T(1) = 1
T(n) = 2T(n-1) + 1

**Iteration Method:**

\[
T(n) = 2T(n-1) + 1 \\
= 4T(n-2) + 2 + 1 \\
= 8T(n-3) + 2^2 + 2 + 1 \\
\ldots
\]

after \( i = n-1 \) steps the iterations will stop

\[
= 2^i T(n - i) + 2^{i-1} + \ldots + 2 + 1 = 2^{n-1} T(1) + \sum_{i=0}^{n-2} 2^i = 2^{n-1} + 2^{n-1} - 1 \\
= 2 \cdot 2^{n-1} - 1 = 2^n - 1
\]

\[
T(n) = \Theta(2^n)
\]

**Question 6: Analysis of the Harmonic series**

T(1) = 1
T(n) = T(n-1) + 1/n

Find \( T(n) = \Omega(?) \)
Solution:

As described in Cormen 3.9 (Approximation by Integrals), we can use an integral to approximate sums:

$$ T(n) = \sum_{i=1}^{n} \frac{1}{i} = 1 + \sum_{i=2}^{n} \frac{1}{i} \leq 1 + \int_{1}^{n} \frac{dx}{x} = 1 + \ln n $$

Question 7: master method example

$$ T(n) = 3T(n/2) + n \log n $$

Use the Master-Method to find T(n).

Solution:

\( a = 3 \), \( b = 2 \), \( f(n) = n \log n \)

We assume it's the 1-st case of a Master Theorem, so we need to show that:

$$ n \log n = O(n^{\log_{2}3 - \epsilon}) $$

\( x = \log_{2}3 = 1.585 \)

\( n \log n \leq cn^{x-\epsilon} \)

\( \log(n) \leq cn^{x-\epsilon-1} \)

We may operate log on both sides (log is a monotonic increasing function and thus we are allowed to do this):

\( \log(n) \leq (x - \epsilon - 1) \log n + \log e = (0.585 - \epsilon) \log n + \log e \)

Next, we need to find values of \( c, \epsilon, n_0 \), such that:

\( \log(n) \leq (0.585 - \epsilon) \log n + \log e \)

Let's choose \( c=1 \): \( \log(n) \leq (0.585 - \epsilon) \log n \)

\( \log(f(n)) \) is smaller then 0.5\( f(n) \) (\( f(n) \) is a monotonous increasing function), for \( f(n) > 4 \).

Thus, we may choose \( \epsilon = 0.085 \).

\( f(n) = \log n = 4 \). Hence we set \( n_0 = 16 \).

\[ T(n) = \Theta(n^{\log_{2}3}) \]

Question 8: analysis of algorithms for multiplication

Given two binary numbers x, y with n-bits each, give an efficient algorithm to calculate the product z = xy.

Analyze the time complexity of this algorithm.
Solution:
Let's look at another problem first:
The mathematician Carl Friedrich Gauss (1777–1855) once noticed that although the product of two complex numbers

\[(a + bi)(c + di) = ac - bd + (bc + ad)i\]

seems to involve four real-number multiplications, it can in fact be done with just three: ac, bd, and \((a + b)(c + d)\), since:

\[bc + ad = (a + b)(c + d) - ac - bd.\]

In our big-O way of thinking, reducing the number of multiplications from 4 to 3 seems wasted ingenuity. But this modest improvement becomes very significant when applied recursively.

Let’s move away from complex numbers and see how this helps with regular multiplication. Suppose x and y are two n-bit integers; and assume for convenience that n is a power of 2.

Method 1: Grade-school Multiplication
Multiplying two numbers of n bits each, in grade-school method, is actually multiplying and n-bit number by 1 bit, n times, and summing up the products. Time: \(\Theta(n^2)\)

Example (no need to get into):

\[
\begin{array}{c c}
10110110 & 182_{10} \\
\times 10011101 & \times 157_{10} \\
\hline
10110110 \\
00000000 \\
10110110 \\
10110110 \\
10110110 \\
00000000 \\
00000000 \\
10110110 \\
\hline
1101111110011110 & 28574_{10}
\end{array}
\]

Method 2: Recursive Multiplication
As a first step towards multiplying x and y, split each of them into their left and right halves, which are n/2 bits long:
For instance, if \( x = 10110110 \) then \( x_L = 1011, x_R = 0110 \), and \( x = 1011 \times 2^4 + 0110 \). The product of \( x \) and \( y \) can then be rewritten as:

\[
x \times y = (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) = 2^n x_Ly_L + 2^{n/2}(x_Ly_R + x_Ry_L) + x_Ry_R.
\]

Let’s analyze the runtime:

- Additions – \( O(n) \)
- Multiplications by powers of two (actually left-shifts) – \( O(n) \)
- Four \( n/2 \)-bit multiplications – \( x_Ly_L, x_Ly_R, x_Ry_L, x_Ry_R \) – with recursive calls.

Our method for multiplying \( n \)-bit numbers starts by making recursive calls to multiply these four pairs of \( n/2 \)-bit numbers (four sub-problems of half the size), and then evaluates the expression above in \( O(n) \) time.

Writing \( T(n) \) for the overall running time on \( n \)-bit inputs, we get:

\[
T(n) = 4T(n/2) + O(n).
\]

Using master method we get \( T(n) = \Theta(n^2) \) – no improvement to gradeschool method!

**Method 3: Improved Recursive Multiplication**

This is where Gauss’s trick comes to mind. Although the expression for \( xy \) seems to demand four \( n/2 \)-bit multiplications, as before just three will do: \( x_Ly_L, x_Ry_R, \) and \((x_L + x_R)(y_L + y_R)\) since

\[
x_Ly_R + x_Ry_L = (x_L + x_R)(y_L + y_R) - x_Ly_L - x_Ry_R.
\]

And the runtime is improved to:

\[
T(n) = 3T(n/2) + O(n).
\]

Again, we use the master method to find \( T(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.6}) \)

The point is that now the constant factor improvement, from 4 to 3, occurs at every level of the recursion, and this compounding effect leads to a dramatically lower time bound.
Pseudo-code:

```plaintext
function multiply(x, y)
    Input: Positive integers x, y, in binary
    Output: Their product

    n = max(size of x, size of y)
    if n = 1: return xy

    x_L, x_R = leftmost \lfloor n/2 \rfloor, rightmost \lceil n/2 \rceil bits of x
    y_L, y_R = leftmost \lfloor n/2 \rfloor, rightmost \lceil n/2 \rceil bits of y

    P_1 = multiply(x_L, y_L)
    P_2 = multiply(x_R, y_R)
    P_3 = multiply(x_L + x_R, y_L + y_R)
    return P_1 \times 2^n + (P_3 - P_1 - P_2) \times 2^{n/2} + P_2
```

Just FYI - We can take this idea even further, splitting the numbers into more pieces and combining them in more complicated ways, to get even faster multiplication algorithms. Ultimately, this idea leads to the development of the Fast Fourier transform, a complicated divide-and-conquer algorithm that can be used to multiply two n-digit numbers in O(n log n) time.

(Question 8 solution adapted from the book “Algorithms” © 2006 by Dasgupta, Papadimitriou and Vazirani
Available online: http://www.cs.berkeley.edu/~vazirani/algorithms.html
And from lesson 1 of http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/ )