# CS 4349 — Advanced Algorithm Design and Analysis, Homework 4

https://iniyanjoseph.com | iniyanijoseph@gmail.com

## Question 1

We must show that $\sum_{k=1}^{n-1} P(k)P(n-k) = \Omega(2^{n-2})$ for $n \geq 5$
We show this by induction.
Before we begin, not that $P(n) = \sum_{k-1}^{n} P(k)P(n-k)$ and $P(1) = 1$. **Proof. Basis:**

- We must show that $\sum_{k=1}^{4} P(k)P(5-k) \geq 2^{5-2}$

$$= P(1)P(4) + P(2)P(3) + P(3)P(2) + P(4)P(1)$$
$$= 2 \cdot P(4) + 2 \cdot P(3)P(2)$$
$$= 2 \cdot (2 \cdot P(3) + P(2)P(2)) + 2 \cdot P(3)P(2)$$
$$= 2 \cdot (2 \cdot P(3) + 1) + 2 \cdot P(3)$$
$$= 2 \cdot (2 \cdot (2 \cdot P(1)P(2)) + 1) + 2 \cdot (2 \cdot P(1)P(2))$$
$$= 2 \cdot (2 \cdot (2 \cdot P(1)P(1)) + 1) + 2 \cdot (2 \cdot P(1)P(1))$$

$= 2 \cdot (2 \cdot 2 + 1) + 2 \cdot 2$

$$= 14 \geq 2^3 = 8$$

**Inductive Hypothesis:**

- Assume that $P(r) = \sum_{k=1}^{r-1} P(k)P(r-k) = P(1)P(r-1) + ... + P(r-1)P(1) \geq 2^{r-2}$ for $r < n$

**Inductive Step**

- We must show that $P(n) = \sum_{k=1}^{n-1} P(k)P(n-k) \geq 2^{n-2}$

By our assumption, $P(n-1) = P(1)P(n-2) + P(2)P(n-3)... + P(n-3)P(2) + P(n-2)P(1) \geq 2^{n-3}$

$$= P(n) = P(1)P(n-1) + P(2)P(n-2) + ... + P(n-2)P(2) + P(n-1)P(1)$$
$$\geq 2 \cdot P(n-1) + P(2)P(n-2) + ... + P(n-2)P(2)$$

By the inductive hypothesis, this means that

$$\geq 2 \cdot 2^{n-3} + P(2)P(n-2) + ... + P(n-2)P(2)$$
$$= 2^{n-2} + P(2)P(n-2) + ... + P(n-2)P(2)$$
$$\geq 2^{n-2}$$

## Question 2

Determine all of the longest common subsequences of the sequences {a, b, b, a, b, a, b, a}, which we call sequence $x$ and {b, a, b, a, a, b, a, a, b}, which we call sequence $y$.
The DP solution to LCS is applicable here. We modify this to maintain a set at every single element of the dp in addition to the length. If we add a new element (i.e. $x_i = y_i$), then we also concatenate $x_i$ to each element in the set as well. Otherwise, if $x_i \neq y_i$, we copy the same set from the dp table which we choose from.
The subsequences are

```
abbaab
ababab
bbabaa
ababaa
bababa
bbabab
abaaba
bbaaba
```

# Question 3

## a

Design recursive and bottom-up dynamic programming algorithms for Tribonacci numbers.
**Recursive approach**
Define table dp
let dp $=< 0, 0, ..., 0 >$ s.t. $|\text{dp}| = n$
We set $\text{Tribonacci}(k) = \text{Tribonacci}(k-1) + \text{Tribonacci}(k-2) + \text{Tribonacci}(k-3)$.
Take $\text{Tribonacci}(i) = \text{dp}[i] \Leftrightarrow \text{dp}[i] \neq 0$, otherwise, compute $\text{Tribonacci}(i)$ using the recursive formula above and set $\text{dp}[i] = \text{Tribonacci}(i)$

```
Input: An integer n
Output: The n_th Tribonacci number
let dp =< 0, 0, ..., 0 > s.t. |dp| = n
dp[1] = 0
dp[2] = 0
dp[3] = 1
function Tribonacci(k)
    if dp[k] ≠ 0
        return dp[k]
    else
        return Tribonacci(k − 1) + Tribonacci(k − 2) + Tribonacci(k − 3)
```

**Bottom Up Approach**
The bottom up approach is defined as follows:
Define table which we call "dp"
let dp $=< 0, 0, ..., 0 >$ s.t. $|\text{dp}| = n$
$\text{dp}[i] = \begin{cases} i \leq 1 \rightarrow \text{dp}[i]=0 \\ i=2 \rightarrow \text{dp}[i]=1 \\ i>3 \rightarrow \text{dp}[i]= \text{dp}[i-1]+ \text{dp}[i-2]+ \text{dp}[i-3] \end{cases}$
Converting this into pseudocode:

```
Input: An integer n
Output: The n_th Tribonacci number
let dp =< 0, 0, ..., 0 > s.t. |dp| = n + 1
for i = 1...(n + 1)
    if i ≤ 2
        dp[i] = 0
    else if i = 3
        dp[i] = 1
    else
        dp[i] = dp[i - 1] + dp[i - 2] + dp[i - 3]
return dp[n+1]
```

## b

Explain the running time of each algorithm.
**Recursive Approach**
The recursive approach takes $O(n)$. Once the table is filled, the algorithm will not recurse further than a single recursive call. Additionally, the table contains only $n$ elements, so the algorithm takes $O(n)$
**Bottom Up Approach** The bottom up approach takes $O(n)$. Filling each entry of the table takes $O(1)$, and there are $n$ entries in the table. Additionally, every element of the table must be enumerated to produce the correct output.

# Question 4

Take the recurrence $T(n) = T(n-1)^3\sqrt{T(n-2)}$, with $T(0) = 2$ and $T(1) = 1, T(2) = 5$.
The naive recursive solution can be implemented as follows:

```
function T(n)
    if n = 1
        return 5
    if n = 0
        return 2
    else
        return T(n − 1)^3 √(T(n − 2))
```

This solution is clearly exponential in $n$, as we must compute the results at every single recursive call, leading to a full tree of recursive calls being generated. The height of this tree is at most $n$, so the number of nodes (i.e. the number of recursive calls) is $O(2^n)$.
Compare this to the bottom-up dynamic programming algorithm for computing the Fibonacci sequence presented below:

```
function T(n)
    let dp =< 0, 0, ..., 0 > s.t. |dp| = n
    for i = 1...n
        if i = 1
            dp[i] = 5
        if n = 0
            dp[i] = 2
        else
            dp[i] = dp[n − 1]^3 √(dp[n − 2])
    return dp[n]
```

Because the dp table contains $n$ elements, so the algorithm takes $O(n)$.