PAL: Tutorial, Sheet 1

Question 1

$a(0) = 0 \qquad a(1) = 1$

$a(n) = a(n-1) + a(n-2) + 1$

Part 1a - Recursive Sequential Algorithm

```
fun calcA(n) {
    if n == 0 then return 0
    if n == 1 then return 1
    return calcA(n - 1) + calcA(n - 2) + 1
}
```

Part 1b - Recursive Parallel Algorithm

```
fun calcA(n) {
    if n == 0 then return 0
    if n == 1 then return 1
    x = spawn calcA(n - 1)
    y = calcA(n - 2)
    sync
    return x + y + 1
}
```

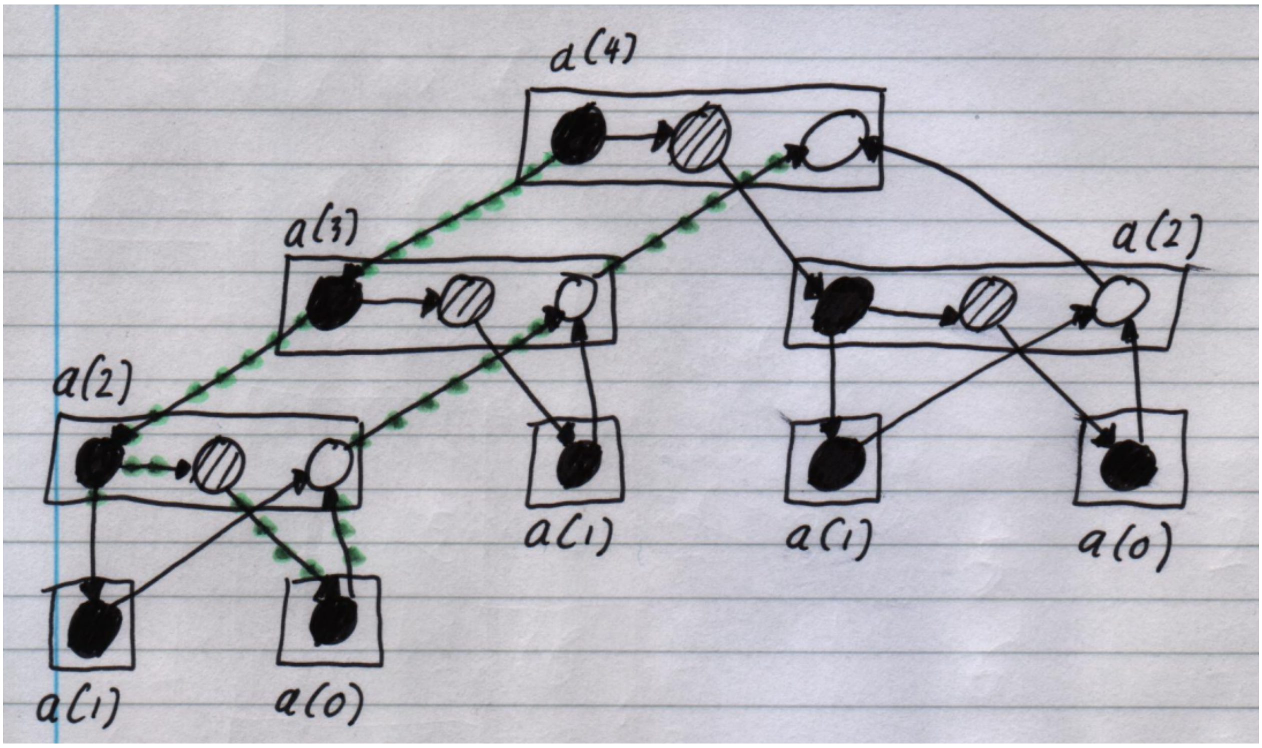
Part 2 - Values

$$\begin{aligned} a(2) &= a(1) + a(0) + 1 \\ &= 1 + 0 + 1 \\ &= 2 \end{aligned}$$

$$\begin{aligned} a(3) &= a(2) + a(1) + 1 \\ &= 2 + 1 + 1 \\ &= 4 \end{aligned}$$

$$\begin{aligned} a(4) &= a(3) + a(2) + 1 \\ &= 4 + 1 + 1 \\ &= 7 \end{aligned}$$

Part 3 - Computation DAG of $a(4)$



Work: 17 calls to `calcA`, **`spawn`** and **`sync`**.

Span: 8 nodes on the critical path (shown in green).

Question 2

$a(0) = 0 \qquad b(0) = 1$

$a(n) = a(n-1) + b(n-1) + 1$

$b(n) = b(n-1) + 2a(n-1) + 1$

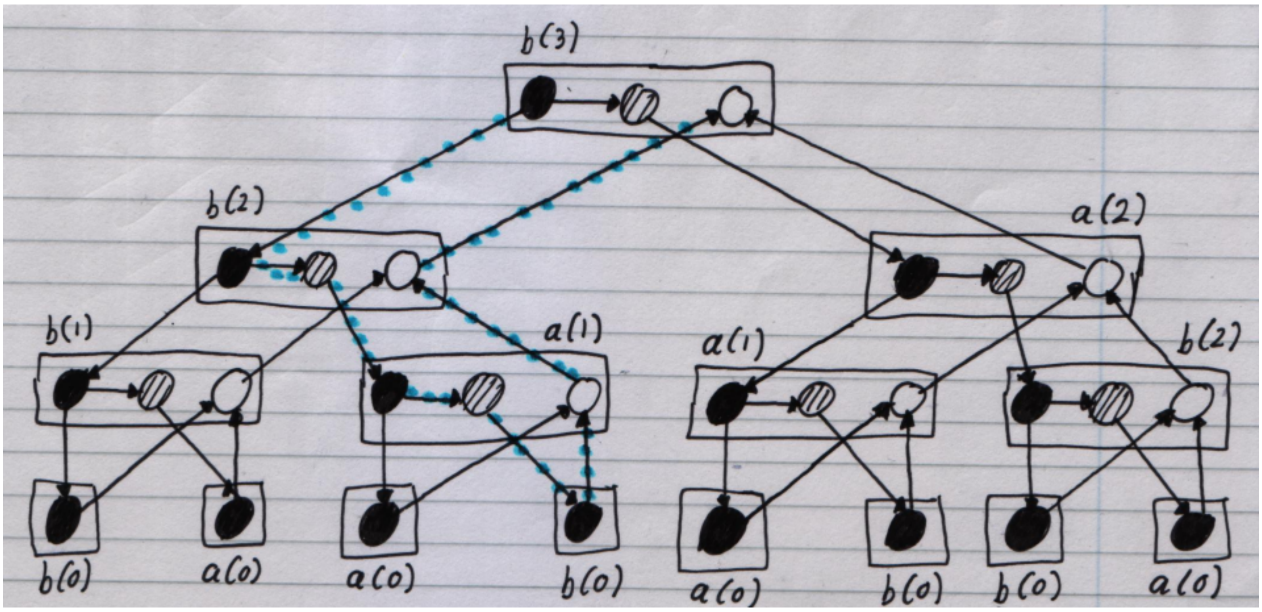
Part 1a - Recursive Sequential Algorithm

```
fun calcA(n) {
    if n == 0 then return 0
    return calcA(n − 1) + calcB(n − 1) + 1
}

fun calcB(n) {
    if n == 0 then return 0
    return calcB(n − 1) + (2 * calcA(n − 1)) + 1
}
```

Part 1b - Recursive Parallel Algorithm

```
fun calcA(n) {
    if n == 0 then return 0
    x = spawn calcA(n - 1)
    y = calcB(n - 1)
    sync
    return x + y + 1
}

fun calcB(n) {
    if n == 0 then return 0
    x = spawn calcB(n - 1)
    y = calcA(n - 1)
    sync
    return x + (2 * y) + 1
}
```

Part 2 - Computation DAG of $b(3)$



Work: 29 calls to `calcA`, `calcB`, **`spawn`** and **`sync`**.

Span: 9 nodes on the critical path (shown in blue).

Question 3

Part 1 - Recursive `MPSum()` for $n = 2^k$

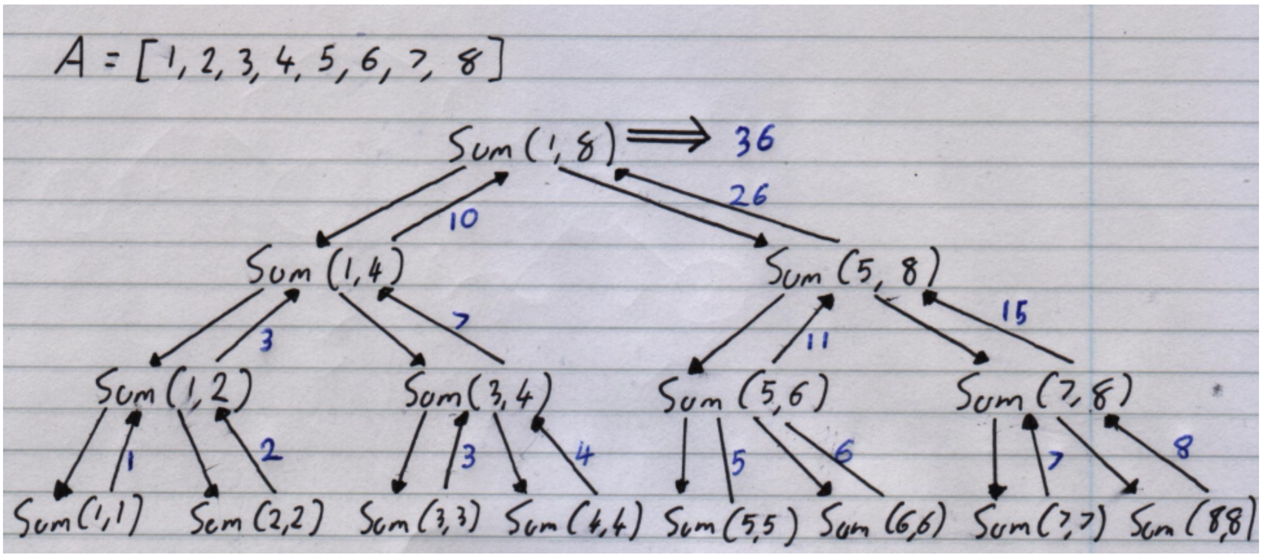
```
fun MPSum(array A) {
    return MPSum(A, 1, A.length)
}

fun MPSum(array A, int s, int e) {
    if s == e then return A[s]

    n = e − s + 1
    x = spawn MPSum(A, s, s + n/2 − 1)
    y = MPSum(A, s + n/2, e)
    sync
    return x + y
}
```

Part 2 - Recursive `MPSum()` for General n

```
fun MPSum(array A) {
    nextPowerOfTwo = 2 ^ roundUp(log_2(A.length))
    return MPSum(A, 1, nextPowerOfTwo)
}

fun MPSum(array A, int s, int e) {
    if s > A.length then return 0
    if s == e then return A[s]

    n = e − s + 1
    x = spawn MPSum(A, s, s + n/2 − 1)
    y = MPSum(A, s + n/2, e)
    sync
    return x + y
}
```

This algorithm pads with zeros up to the next power of two. The $O(log_2(n))$ performance is not affected by this approach.

Part 3 - Execution for $A = [1, 2, 3, ..., 8]$



Work for $A[1..n]$ is $O(2^{k+1})$ if $n = 2^k$.

Span for $A[1..n]$ is proportional to $log_2(n)$.

Part 4 - Recursive `MPMax()` for $n = 2^k$

```
fun MPMax(array A) {
    return MPMax(A, 1, A.length)
}

fun MPMax(array A, int s, int e) {
    if s == e then return A[s]

    n = e − s + 1
    x = spawn MPMax(A, s, s + n/2 − 1)
    y = MPMax(A, s + n/2, e)
    sync
    if x > y
        return x
    else
        return y
}
```

Part 5 - Recursive `MPMember()` for $n = 2^k$

```
fun MPMember(array A, int x) {
    return MPMember(A, x, 1, A.length)
}

fun MPMember(array A, int x, int s, int e) {
    if s == e
        if A[s] == x
            return true
        else
            return false

    n = e − s + 1
    x = spawn MPMember(A, s, s + n/2 − 1)
    y = MPMember(A, s + n/2, e)
    sync
    return x OR y
}
```