THE UNIVERSITY OF SYDNEY

# COMP2129 Week 12 Tutorial

**Recursion, $p$-leaf trees, and Barriers**

## Exercise 1: Recursion Overhead

Some functions or algorithms are much more easily thought about as being *recursive*, in that they call themselves, with a smaller version of the problem. This is great for describing algorithms, but sometimes a bad way of implementing them. For example in C, every function call pushes a new *stack frame* onto the program stack. This stack frame is very large (at least in comparison to the bare minimum information needed to recurse), and if too many frames get pushed on, the stack will overflow and a segmentation fault will occur.

A string is a *palindrome* if it reads the same forwards and backwards. For example, the strings `"anna"`, `"madam"`, and `"racecar"` are all palindromes. If we define the empty string `""` to also be a palindrome, this leads to a very nice recursive algorithm, by repeatedly chopping the ends of the string off.

```
1 int is_palin(char *str, int len) {
2     if (len <= 1) {
3         return 1;
4     }
5     if (str[0] != str[len-1]) {
6         return 0;
7     }
8     return is_palin(str+1, len-2);
9 }
```

1. Draw on a piece of paper the recursive calls the algorithm makes to check the strings `"anna"`, `"racecar"`, and `"comp2129"`.
2. Copy the function into a C file, and make sure it works by running it on some test strings.
3. Create a very long ($\approx 100,000$ characters) palindrome. Check the algorithm works. How long can you make the string until you get a segmentation fault?
4. Write an iterative version of the same function, by putting the function body inside a loop, and replacing the recursive call at the end with just a reassignment of the variables `str` and `len`. Ensure that it passes all of your previous tests, and does not cause a stack overflow for large strings.
5. See if you can test the speed of the recursive and iterative functions. How much faster is the iterative version?

# Exercise 2: $p$-leaf tree

The following pieces of pseudocode show different methods of using recursion in parallel using threads. The number of threads created is bounded, through the depth variable. You can assume that the **`recurseFunc`** method is always called using **`depth = 1`** as an argument, and **`maxdepth = 4`** always.

Version 1 of the recursion always creates new threads to recurse.

```
1  // Version 1.
2  recurseFunc(array, depth) {
3      // left, right = an even split of array into two halves.
4      if (depth < maxdepth ) {
5          A = createThread(recurseFunc, left, depth + 1);
6          B = createThread(recurseFunc, right, depth + 1);
7          join(A);
8          join(B);
9      } else {
10         recurseFunc(left);
11         recurseFunc(right);
12     }
13     combine(left, right);
14 }
```

Version 2 of the recursion creates one new thread, and uses the current thread, to recurse.

```
1  // Version 2.
2  recurseFunc(array, depth) {
3      // left, right = an even split of array into two halves.
4      if (depth < maxdepth ) {
5          A = createThread(recurseFunc, left, depth + 1);
6          recurseFunc(right);
7          join(A);
8      } else {
9          //normal recursion
10         recurseFunc(left);
11         recurseFunc(right);
12     }
13     combine(left, right);
14 }
```

1. Draw call trees for both versions 1 and 2, labelling each node with what thread is executing that node. Number the threads $T_1$, $T_2$, and so on.
2. How many threads does version 1 create? How many threads does version 2 create?
3. How many threads are left idle in each version? Supposing each thread is directly mapped to a processor, which version makes better use of available resources?

Version 3 of the code uses a different approach. It uses a condition variable **`active_threads`**, which holds the number of currently active threads. It also has some **`max_threads`** variable, rather than just a **`max_depth`** variable. You can assume that there are no race conditions in the code.

```
1  // Version 3.
2  recurseFunc(array, depth) {
3      // left, right = an even split of array into two halves.
4      if (active_threads < max_threads) {
5          A = createThread(recurseFunc, left, depth + 1);
6          active_threads++;
7          join(A);
8          active_threads--;
9      } else {
10         recurseFunc(left);
11     }
12
13     if (active_threads < max_threads) {
14         B = createThread(recurseFunc, right, depth + 1);
15         active_threads++;
16         join(B);
17         active_threads--;
18     } else {
19         recurseFunc(right);
20     }
21
22     combine(left, right);
23 }
```

1. How much more complicated is this to implement than versions 1 and 2?
2. Draw one possible call tree, with labelled threads.
3. What are the advantages of using version 3 over 1 and 2? (Hint: what if some parts of the array take longer to process than others?)

# Exercise 3: Barriers - Cellular Automata

Cellular automata are tiny models of life, made up of cells on an $n$-dimensional grid, where each cell is either dead or alive (more complicated models are also possible). Each configuration is called a *generation*, with deterministic rules on how to proceed to the next generation. A famous example of this is Conway's Game of Life.

In this question, we'll consider a simple example, where the cells live on a one-dimensional line. To check if cell $[i]$ is alive in the next generation, we check its ancestors $[i-1]$ and $[i+1]$ in the current generation. If neither ancestor is alive, the cell dies. If one (but not both) ancestors are alive, the cell becomes alive. If both ancestors are alive, the cell dies (through overcrowding). The rules are visually shown in Figure 1.

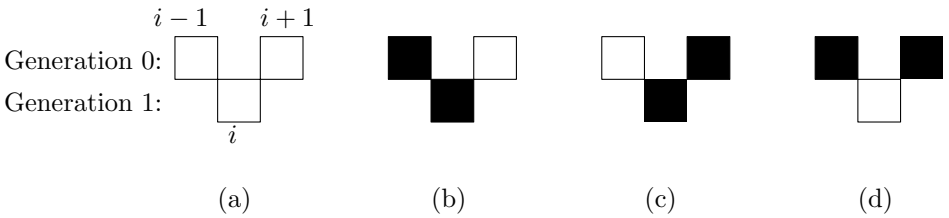


Figure 1: (a): The cell cannot be born without an ancestor. (b) and (c) show the cell being born from one ancestor. (d) shows the cell dying through overcrowding.

If we represent the cells as an array of integers, with a dead cell represented by $0$ and a live cell represented by $1$, then the next cell's state is just the XOR of its ancestors, which we can do by using the C operator **`a^b`**.

The file **`automata.c`**, available on eLearning and Piazza, gives an implementation of this cellular automaton. It starts with a generation with only one live cell in the middle of the board, and then loops, on each loop printing out the current generation and then computing the next. It prints out a `.` for a dead cell, and an **`X`** for a live cell.

1. Download, compile, and run the file. Do you notice any patterns?
2. Create a worker function (suitable to be passed in to **`pthread_create()`**) which implements the **`next_gen()`** function for a slice of the array. Then replace where **`next_gen()`** is called in the main loop by calling your worker function once.
3. Create a new variable **`int nthreads = 4`**. Replace the single call of your worker function with **`nthreads`** number of calls to your worker function, each working on a fraction of the array.
4. Now instead of just calling the worker functions, use **`pthread_create()`** and **`pthread_join()`** to create and collect threads for each generation.
5. Refer back to the last tutorial and the lecture slides on how to use barriers. The goal is now to only call **`pthread_create()`** four (or **`nthreads`**) times in the whole program, and use a barrier to synchronise between threads. Each worker thread should compute its portion of the next generation, then sit waiting for the main thread to finish printing out the generation before they can start again.