Name: _____

IB Computer Science 2
Call stack worksheet

Download `StackPointer.zip` to the folder where you usually write activity code, and extract it. Run the command `xattr -d com.apple.quarantine libStackPointer.dylib` so macOS trusts it. You now have access to a `StackPointer.getStackPointer()` method, which returns the **memory address of the calling method's local variables** as a `long`.

1. Start a new program that prints out `main()`'s stack pointer. What do you notice when you run it multiple times?

2. Store `main()`'s stack pointer in a `static` variable. *Without declaring any local variables*, write and call a helper method that prints the difference between the two methods' stack pointers. What is the memory cost of a method call, in bytes? Where are the callee's variables stored in memory relative to those of its caller?

3. Make a copy of the helper method, and define exactly two local `int` variables. Looking at the *additional* memory cost of calling this method, what is the in-memory size of an `int`, in bytes?

4. Look up the size of a Java `int`. How much memory overhead does the JVM impose to store each `int`? (That is, how much space is wasted?)

5. Do you expect reference types to take more or less space, and why? Test your prediction by changing the type of both local variables. What is the size of each reference type?

6. What relationship do you notice between the in-memory size of the two types? Why do you think the JVM authors opted to store `int`s inefficiently?

(over)

7. Write a recursive method that takes a single `int` parameter, the number of times it should call itself. Each invocation should print the difference from `main()`'s stack pointer, then recurse unless it has reached the base case. Call the method and write down the memory overheads of its first few recursive calls.

8. Why can't there be a single address corresponding to the declaration of a local variable in the source code, like for `static` variables? Looking at the memory overheads you recorded and thinking about how they were computed, why is it called the call *stack*?

9. In `main()`, change the argument so that your function will recurse 1000 times. Approximately how much total memory overhead do you expect, in bytes? What happens when you run the program?

10. At the beginning of the recursive method, define a local array of 10 million `int`s. What happens when you run the program, and why?

11. Scroll up and compare the memory overheads of the first few recursive calls against the ones you recorded above. What do you notice? Where are(n't) the arrays stored?

If you have extra time…

You may have noticed that the overhead of the first call to your recursive method matched that of your non-recursive method. This might be surprising given that the recursive one has a single local variable (its parameter) but the other has two. In the non-recursive one, define only one `int` variable and record the additional overhead compared to your answer to (2). Add a second variable and repeat. Continue this a few times and make a table. Work out the relationship. Now start over with `byte` variable(s), comparing against your table. What happens and why?