ECSE 324 - Lab 2 Report

Alexander Liu

February 23rd, 2024

Part 2: Ring my Pell

Iterative Approach

The function only intakes a single input with is held in the register A1. I used registers A2-A4 and V1 to store all the relevant variables that are needed within the function. This includes the seeds (or the base cases 0 and 1), the summation, and the index for the loop. Note that the index starts at 2 since we already have our 2 base cases and only need to start at the 3rd iteration which corresponds to index 2. The Pell function is a subroutine, so we must push the link register, V1, and V2 onto the stack at the beginning of the function and pop them at the end when we are done, and the final value is moved into A1. Then, we simply run a BX LR to return to wherever the PC was in the caller.

Performance wise, the iterative approach runs in linear time which means that the number of executed instructions scales linearly as the input *n* increases. To be more specific, I will be conducting an analysis on my implementation. If we have base case $n = 0$, then a total of 13 instructions are executed. If we have base case $n = 1$, then a total of 17 instructions are executed. For any arbitrary input *n* greater than 1, the number of instructions executed would be $17 + (n-1) * 8$ (we execute 11 instructions to get to the loop body; we execute the loop body containing 8 instructions (n-1) times; we execute the compare and branch instructions on the (*n*)th iteration; we execute 3 instructions in the *_end* block; we execute the last instruction in *_start* when we branch back). In terms of memory accesses, the iterative implementation is efficient in the sense that it does not need to store or load anything from memory, except for the link register and fetching instructions. In terms of code size, the iterative code is 112 bytes.

Recursive Approach

The recursive approach is much more complex. Firstly, there are a lot more interactions with the stack. Since we are running the function recursively, then we must push all the relevant values onto the stack before stepping into the recursive call. This includes registers V1, V2 which store the values for n-1 and n-2 respectively as well as the link register. Due to the nature of how Pell numbers are calculated, two recursive calls are required: one for *n*-1 and one for *n*-2. Therefore, after the first recursive call, we must push the result onto the stack and run all the operations for *n*-2. Then, we must pop the result for *n*-1 from the stack and do the necessary arithmetic. We push the link register onto the stack at the beginning of each call of the function to be able to backtrack to where the caller was after all the recursive steps are complete. The relevant BX LR instruction in the *_end* block leads the program to wherever the caller was when the callee Pell was called.

Performance wise, the recursive version is a lot more resource intensive than the iterative version. Firstly, the subroutine's code size is 128 bytes. The recursive implementation also executes many memory accesses. This can be attributed to the fact that we constantly need to push and pop from the stack. Finally, the number of instructions executed scales exponentially as *n* increases. This is because for each function call, there are two subsequent recursive calls which means that the number of total calls doubles as we go down a layer. Overall, we can use $\sim (2\text{^}n) * 29$ to establish an upper bound and estimate the number of instructions executed given an input *n* since the subroutine is 29 instructions long.

Observations

In terms of code size, the recursive implementation is slightly larger. Even though the difference is not remarkable, the difference in size can be attributed to the fact that we are making 2 recursive calls and that we need to push and pop register values. In terms of number of instructions executed, the iterative version is vastly more efficient than the recursive one as its number of instructions executed scales linearly rather than exponentially. It is worthy of noting that for very small *n*, both implementations will have about the same number of instructions executed. However, as *n* increases, the iterative implementation becomes much more advantageous. In terms of memory accesses, the iterative version also comes out on top as it does not have to push or pop anything to the stack except for the link register when it is called. As memory accesses are a lot slower than storing and loading values from the register file, this would mean that the recursive implementation's runtime will also suffer in that regard. The table below offers concrete data to visualize these trends.

Input n	Iterative			Recursive		
	# Instructions Executed	# Memory Accesses	Output	# Instructions Executed	# Memory Accesses	Output
0	13	21	0	12	20	0
5	49	57	29	212	318	29
10	89	97	2,378	2,542	3782	2,378
25	209	217	1,311,738,121	3,491,722	5,191,218	1,311,738,121
26	217	225	3,166,815,962	5,649,736	8,399,582	3,166,815,962

**Note: For the purposes of this report, fetching an instruction is counted as a memory access.*

Part 3: Sort It

Bubble Sort

The function has two inputs which are the address of the array (i.e. a pointer to the first element of the list) as well as the length of the array. Bubble sort is a straightforward sorting algorithm that compares each element of the array to every subsequent element and swaps them if the latter is less than the former. In practice, bubble sort steps through the list, compares adjacent elements, and swaps them if they are in the wrong order until the entire list is sorted. Since we do not do anything after calling the function *sort*, we do not need to push the link register. Lastly, my implementation of bubble sort sorts the array in place to minimize memory usage. I use a register to store a temporary variable when swapping 2 elements.

The function takes up a total of 128 bytes. The number of instructions executed is expected to grow exponentially as the length of the input array increases as we have 2 loops inside the function body. The outer loop is called n-1 times while the inner loop is called n-i-1 times for each ith iteration of the outer loop, meaning that the number of times the inner loop decreases as the index i of the outer loop increases. For the purposes of this analysis, we shall take the worst-case scenario of the inner loop to gauge an upper bound to how many instructions are executed. This worst-case scenario would be at index i=0 where the inner loop would have to run n-1 times. Therefore, we can say that the number of instructions executed is ~ (n-1) * 9 + (n-1) ^ 2 * 13 as the outer loop is 9 instructions long and the inner loop is 13 instructions long. This would indicate that the number of instructions executed scales quadratically as the length of the input array increases (this formula sets an upper bound, which means that in many cases, the actual number of instructions executed will be less). Finally, in terms of memory accesses, the implementation contains 2 load and 2 push instructions which result in extra memory accesses. It is important to note that the store instructions are not always executed if the condition of *array[j] > array[j+1]* is not met. This means that the quantity of memory accesses really depends on the input array. The worst-case scenario would be an array sorted in decreasing order.

Insertion Sort

Once again, the only inputs to the function are the address of the first element of the array to be sorted as well as its length. We instantiate a variable in register A3 to represent the length of the partition of the array that is sorted which is initially set to 1. Then, we fetch the key which is the element to be inserted into our sorted partition. We use a while loop to shift all the elements that are greater than the key to free a slot for the key. After the while loop, we set the slot's value to the key and increment the length of the sorted partition by 1. Once again, my implementation of insertion sort sorts the array in-place to conserve memory.

The function implementation takes up a total of 112 bytes (the global variables are identical to the ones in bubble sort for a fair comparison). The outer loop labeled executes *n-1* times where *n* is the length of the input array since we start iterating at index 1. The number of instructions executed in the inner while loop will depend on how sorted the input array is. In the worst-case scenario, the input array would be sorted in decreasing order which means that the inner loop will have to shift every single element in the sorted partition for every element we try to insert. To have an equitable comparison to bubble sort, I will use the worst-case scenario for the inner loop. This worst-case scenario occurs when we must insert the last element in a decreasingly sorted array and must shift the n-1 elements in the sorted partition to insert this last element. Assuming this worst-case scenario, we would have a total amount of ~ (n-1) * 9 + (n-1) ^ 2 * 9 instructions to execute since the *_forLoop* body contains 9 instructions and the *_whileLoop* body contains 9 instructions. In a similar fashion, the number of memory accesses would also depend on the input array. This means that given the worst-case of a decreasing array, the subroutine would run a lot of memory accesses to shift the elements in the sorted partition. Conversely, if the array is partially sorted already, then the number of memory accesses is reduced.

Observations

The difference between the code size of the two sorting functions is trivial. Both functions have two loops which leads to them being similar in size. In terms of instructions executed, insertion sort will execute a similar number of instructions to bubble sort in the worst-case scenario of a decreasingly sorted array. However, insertion sort will execute less instructions for inputs that are already partially sorted. This is because bubble sort compares each element to every subsequent element regardless whereas insertion sort's inner loop will typically perform less swaps and comparisons. Consequently, the number of memory accesses insertion sort performs will be less than bubble sort as we are swapping less. Finally, the space required in memory for both methods are the same as they both sort in-place. The table provided below offers concrete data to visualize the described observations.

Input array	Bubble Sort (In-place)		Insertion Sort (In-place)	
	# Instructions Executed	# Memory Accesses	# Instructions Executed	# Memory Accesses
Single element: {1}	8	10	6	8
Partially sorted: {6, 1, 2, 3, 5, 4}	204	248	127	155
Positive & Negative: {1, -1, 2, -2, 3, -3}	222	272	148	180
Worst-case: {8, 7, 6, 5, 4, 3, 2, 1}	449	563	335	407
All identical: {1, 1, 1, 1, 1, 1, 1, 1}	281	339	104	127