Daniel Brown

12/11/20

CIS4930

Project Report

Overview of Problem

My project was about Syntax-Guided Synthesis. Syntax-Guided Synthesis is the problem of generating code from a specification that is given to the computer. It works by creating an outline of a program that has holes in it. The outline can be thought of as the structure of the program; the programmer puts in control flow elements and variable declarations, along with any other programming constructs that are needed, to create a husk of a program. But instead of writing out every line of the program, the programmer leaves holes in. These holes can be thought of as the finer details of the program. For instance, a programmer might create an outline that contains an if statement, but leave a hole in the condition of the if statement, that the computer can fill in. Once a programmer has created an outline with holes, they then use assertions to define what the output of the program should be and how it should operate. The computer will try to fill in the holes in such a way that all of the assertions are always fulfilled, no matter what the input to the program is.

My project specifically was focused on creating outlines for several different sorting algorithms, and then compare the performance of a tool to generate those programs for different sizes of an input array. For my project I chose to create outlines for selection sort, insertion sort, bubble sort, merge sort, and quicksort.

In this report, I will describe the tool that I used to synthesize the sorting algorithms, go over the outlines that I created for each sorting algorithm, talk about the performance of the tool I used on each sorting algorithm, and finally, discuss some observations I had about the synthesized code output.

Sketch

The tool that I decided to use to synthesize the sorting algorithms is called Sketch. Sketch is an imperative programming language created in 2005 that is somewhat similar to C. It's defining features are several programming constructs the language has that allow it to perform program synthesis.

The first of these new constructs is the ?? operator. The ?? operator adds a simple integer hole to the program. When sketch synthesizes a program from a sketch, it will attempt to fill in each ?? operator with a non-negative integer value, such that the value chosen for the ?? operator causes all assertions in the program to always be true. If sketch cannot find a value for ?? that works, it will return an error and not output any synthesized code. For example, in the following code example, we define a variable t that is equal to x times a ?? hole. Then there is an assertion that t is equal to $5 * x$. When synthesizing this program, sketch attempts to find a value for this ?? hole so that the following assertion is true. In this simple case, it is easy to see that the value of this ?? should be 5. The harness keyword next to the function name simply tells sketch that this function is the entry-point to begin attempting to synthesize code.

```
harness void main(int x) {
      int t = x * ??;
      assert(t == 5*x);
}
```

?? Example

```
void _main (int x)
{
      assert((x * 5) == (5 * x));
}
```

?? Example Output

As you can see in the output of this program, sketch automatically filled in the ?? hole with a value of 5, and replaces the variable t in the assertion with the synthesized code for what t is equal to, $x * 5$. Although this is example is incredibly simple, it shows off a very powerful feature of sketch.

In the last example, sketch will always fill in the hole with the value 5. However, it's important to note that there is a random element to how sketch searches for values for holes that work, and this means that sketch will not always produce the same output code for the same input sketch. For instance, in the next example there are several different possible outputs that sketch could produce.

```
harness void main() {
      int t = ?? + ??;
      assert(t == 10);
}
```

?? Example with Multiple Outputs

In this example, there are many different values that sketch could fill in the holes with that would fulfill the assertion. For instance, filling in the first hole with a value of 0 and the second with 10, or by filling in the first hole with a value of 7 and the second with 3.

The other sketch code synthesis feature that I used in this project was its Regex generators. These allow you to write out small bits of code in a very simple regex, that sketch can then use to generate code to fill in a hole. The format for a regex generator in sketch is

{| REGEX |}

Where regex is a regex formula that sketch can parse. In the regex you can write out small pieces of code, use the | character to give sketch a choice between two or more options, use the ? character to make a given section as optional, and use parenthesis to denote order of operations inside the regex. You can also put ?? holes in the regex. This gives the programmer a very powerful tool to define a range of possible code segments for sketch to choose from when synthesizing code.

```
harness void main(int x, int y) {
      int t = {| x (+ | - | *) (y (+|-) ??) |};
      assert(t == x*(y-5));
}
```

Regex Generator Example

```
void _main(int x, int y) {
      assert((x*(y-5)) == (x*(y-5)));
}
```

Regex Generator Example Output

In this example, the regex generator is telling sketch to choose some operator, + - or *, to be performed between x and y plus or minus some number ??. The assertion tells sketch to make this value equal to x*(y-5). You can see that in the output, sketch has done just this, choosing the * operator, and picking the number 5 for the ?? hole to be subtracted from y.

Regex generators in sketch are very powerful if the programmer knows generally what kind of code will fill in a hole, but not the specifics. When this is the case, the programmer can define a large number of possible options very easily, and let sketch figure out what combination of choices causes all assertions to be true

There are a few more code synthesis features of sketch, however I will not go over them because I did not use them in my code.

Sketch also has many different command line options, however I will only mention the 4 that I used in this project

--bnd-unroll-amnt X: This determines how many times loops should be unrolled to evaluate them. This is always equal to the limit value I define in the main function

--bnd-inline-amnt X: This determines how many times functions should be inlined to evaluate them. This is only used for merge and quick sort, as those are the only 2 recursive sorting algorithms I implemented.

--fe-output-code and --fe-output-test: These options cause sketch to output C++ code of the output, as well as a test function to run the code and ensure that it works. I will talk more about these later in the report.

My Code

Next, I will be going over the sketch code that I wrote for each of the different sorting algorithms and explaining the different holes in them. First is the main function.

```
1 harness void main(int n, int[n] in, int y) {
2     int limit = 8;
3     if(n != limit) {
4           return;
5     }
6     if(y < 0 || y >= n - 1) {
7           return;
8     }
9     //in = selection(limit, in);
10    //in = merge(limit, in);
11    //in = quick(limit, in);
12    //in = insertion(limit, in);
13    in = bubble(limit, in);
14    assert(in[y] <= in[y+1]);
15 }
```

Main Function

The inputs for the main function are n, the size of the array to be sorted, in, the array that we need to sort, and y, which is a dummy variable used in the to write the assertion that the array is sorted. Line 2 defines the limit variable, which is how many times sketch should unroll any loops it encounters. Lines 3 through 5 the variable n is always equal to the limit. This is needed because if the size of the array is greater than the limit we defined, then sketch won't be able to unroll the loops in the sorting algorithms all the way to verify that the code it produced works. Lines 6 through 8 ensure that the dummy variable y is within the bounds of the array. After this, the programmer can select which sorting algorithm to run by uncommenting the needed line. The assertion at the end guarantees that the array is sorted by asserting that for any value in the array at position y, the value at position $y + 1$ will be greater than or equal to the last value.

First, I will go over my sketch of the selection sort algorithm.

```
#define CMP {| (CMPSRC (== | < | > | !=) CMPSRC )|}
1 int[n] selection(int n, int[n] in) {
2     int i = 0;
3     while({| i (< | > | == | !=) (((n (+|-) (0|1)))) |}) {
4           int j = i+1;
5           int jmin = i;
6           while({| j (< | > | == | !=) (((n (+|-) (0|1)))) |}) {
7                 #define CMPSRC {| (in[INDEX]) |}
8                 #define INDEX {| (j|jmin) |}
9                 if(CMP) {
10                      jmin = j;
11                }
12                #undef CMPSRC
13                #undef INDEX
14                j = {| j (+|-) ?? |};
15          }
16          if({| jmin (== | < | > | !=) i |}) {
17                int temp = in[jmin];
18                in[jmin] = in[i];
19                in[i] = temp;
20          }
21
22          i = {| i (+|-) ??|};
23    }
24    return in;
25 }
```

Selection Sort

The first hole in this algorithm is at line 3. Here, sketch chooses some comparison between the index variable i and the size n, plus or minus either 0 or 1. The output from this hole determines when the outer loop finishes running. The next hole is at line 6, and is the same as the last hole, but instead of using the index variable i, it uses the index variable j, for the inner loop. At line 9, sketch fills in another hole represented by the macro CMP. This is a macro that I put at the top of the file so that all sorting algorithms I sketched could use it, but I have simply copied it above the function for convenience. This macro defines a regex generator that chooses some sort of comparison between 2 other macros, CMPSRC. In this case CMPSRC is simply the value in[INDEX], where INDEX can be either j or jmin, variables representing the minimum values that the sorting algorithm has found. In different sorting algorithms, I define CMPSRC and INDEX to be different values based on the names of the variables in the other algorithms. The next hole is at line 14, which sketch fills in to either increment or decrement the index j by some value. This same hole is present at line 22, but with the index i. Finally, there is another comparison hole at line 16, which determines if 2 values should be swapped or not in the array.

Next, I will go over my sketch of the insertion sort algorithm.

```
1 int[n] insertion(int n, int[n] in) {
2     int i = 1;
3     while({| i (< | > | == | !=) (((n (+|-) (0|1)))) |}) {
4           int j = i;
5           while({| (j (< | > | == | !=) ??) |} && {| (in[j(+|-)(0|1)] (< | >
| == | !=) in[j(+|-)(0|1)]) |}) {
6                 int temp = in[j-1];
8                 in[j-1] = in[j];
9                 in[j] = temp;
10                j = {| j (+|-) 1 |};
11          }
12          i = {| i (+|-) 1|};
13    }
14    return in;
15 }
```

Insertion Sort

The holes in insertion sort are somewhat similar to the holes in selection sort, however due to the nature of insertion sort, there are fewer holes overall. The first hole at line 3 chooses a comparison between the index i and n plus or minus either 0 or 1, to determine if the sorting algorithm should keep running. At line 5, there are 2 holes in the condition of the while loop to determine if 2 values should be swapped. First sketch chooses some comparison between j and some number ??, then sketch choose another comparison between the array value at some index j plus or minus either 0 or 1 to another array value at index j plus or minus either 0 or 1. Then at lines 10 and 12 sketch fills in the holes to either increment or decrement the index variables i and j.

Next, I will go over my sketch of the bubble sort algorithm.

```
#define CMP {| (CMPSRC (== | < | > | !=) CMPSRC )|}
1 int[n] bubble(int n, int[n] in) {
2     int swapped = ??;
3     while(swapped == ??) {
4           swapped = ??;
5           int i = 1;
6           while({| i (< | > | == | !=) (((n (+|-) (0|1)))) |}) {
7                 #define CMPSRC {| (in[INDEX]) |}
8                 #define INDEX {| i (+|-) ?? |}
9                 if(CMP) {
10                      swapped = ??;
11                      int temp = in[i-1];
12                      in[i-1] = in[i];
13                      in[i] = temp;
14                }
15                #undef CMPSRC
16                #undef INDEX
17                i = {| i (+|-) ??|};
18          }
19    }
20    return in;
21 }
```

Bubble Sort

At lines 2, 3, 4, and 10, sketch fills in the ?? holes with 2 distinct dummy values, one representing a state in which the loop should continue running, and the other representing a state in which the array has been sorted and should finish running. At line 6, sketch makes a comparison between the index i and the array length n, plus or minus either 0 or 1, to determine if the inner loop should continue or not. At line 9 we again make use of the CMP macro, just like in selection sort. However, in this case, the only value that INDEX can be is i plus or minus some number ??. This determines if 2 values should be swapped or not. Finally, at line 17 sketch fills in the hole to either increment or decrement the index i by some amount.

Next, I will go over my sketch of the merge sort algorithm.

```
1 int[n] merge(int n, int[n] in) {
2     if({| n (> | < | == | !=) ?? |}) {
3           return in;
4     }
5     int lsize = n/2;
6     int rsize = n/2;
7     if({| n % 2 (> | < | == | !=) ?? |}) {
8           rsize++;
9     }
10    int[n] left;
11    int[n] right;
12    left[0::lsize] = merge(lsize, in[0::lsize]);
13    right[0::rsize] = merge(rsize, in[lsize::rsize]);
14    int li = 0;
15    int ri = 0;
16    int i = 0;
17    while({| i (< | > | == | !=) (((n (+|-) (0|1)))) |}) {
          if({| ri (< | > | == | !=) rsize |} || ({| li (< | > | == | !=) lsize |}
&& {| (left[li] (== | < | >) right[ri]) |})){
18                in[i] = left[li++];
19          }
20          else {
21                in[i] = right[ri++];
22          }
23          i = {| i (+|-) ??|};
24    }
25    return in;
26 }
```

Merge Sort

Because merge sort is a recursive sorting algorithm, the first hole at line 2 makes a comparison between the array size n and some number ?? to determine if the current array is too small to be broken down again and should instead be returned. The hole at line 7 determines if the number of items being sorted is odd, and if so, adds an item to the right-side array. Then, at line 17, there are 3 holes to determine which subarray to sort a value into the main array from. The first 2 holes in this line are comparisons between the right index and the right array size, and the left index and the left array size, used as bounds checks so that there are no array out of bounds errors. Then, the 3rd hole in this line creates a comparison between the left and right sub arrays to determine which subarray to sort a value into the main array from. Finally, the hole at line 23 increments or decrements the index i by some amount.

Finally, I will go over my sketch of the quick sort algorithm

```
#define CMP {| (CMPSRC (== | < | > | !=) CMPSRC )|}
1 int[n] quick(int n, int[n] in) {
2     if({| n (> | < | == | !=) ?? |}) {
3           return in;
4     }
5     int pivot = in[n-1];
6     int lsize = 0;
7     int rsize = 0;
8     int[n] left;
9     int[n] right;
10    int i = 0;
11    #define CMPSRC {| (pivot | in[i]) |}
12    while({| i (< | > | == | !=) (((n (+|-) (0|1)))) |}) {
13          if(CMP){
14                left[lsize] = in[i];
15                lsize++;
16          }
17          else {
18                right[rsize] = in[i];
19                rsize++;
20          }
21          i = {| i (+|-) ??|};
22    }
23    #undef CMPSRC
24    if({| lsize (> | < | == | !=) 0 |}) {
25          in[0::lsize] = quick(lsize, left[0::lsize]);
26    }
27    in[lsize] = pivot;
28    if({| rsize (> | < | == | !=) 0 |}) {
29          in[lsize+1::rsize] = quick(rsize, right[0::rsize]);
30    }
31    return in;
32 }
```

Quick sort

Similar to merge sort, the first hole in quick sort, at line 2, checks if the current array should be returned, due to the recursive nature of quick sort. The hole at line 12 generates a comparison between the index i and the array size n, to determine if the loop should keep running. At line 13, we again use the macro CMP to determine which subarray the current value should be put into. In this case, the 2 values that are being compared are either pivot or in[i]. The hole at line 21 increments or decrements the index i. Finally, the holes at lines 24 and 28 compare the sizes of the subarrays to the number 0 to determine if quick sort should be recursively called on the subarrays.

Results

Selection Sort

n	Running Time(ms)
2	5756
3	12639
4	41696
5	32924
6	52624
7	153939
8	195027
9	1460370

Insertion Sort

n	Running Time(ms)
2	3846
3	4084
4	4464
5	5732
6	11762
7	11891
8	44295
9	266049
10	3495745

Bubble Sort

n	Running Time(ms)
2	4701
3	13243
4	73773
5	62744
6	38282
7	51082
8	146236
9	372058
10	2443394
11	5759765

Merge Sort

n	Running Time(ms)
2	1003
3	1988
4	2443
5	2656
6	4845
7	3771
8	2017
9	6358
10	8225
11	8243
12	8936
13	23068
14	211680
15	9880
16	149302

Quick Sort

n	Running Time(ms)
2	1801
3	5377
4	5021
5	11874
6	38863
7	122917
8	712275

Observations

As you can tell, for all but merge sort, the running time increased roughly exponentially with the size of n. I will explain what was weird about merge sort soon. For each of the sorting algorithms, the highest value of n I have an observation for was the highest value of n I could get an output of. For selection insertion and bubble sort, trying to run them with a value of n 1 above their current maxes led to the program running for too long to reasonably wait for. However, for both merge and quick sort, trying to run them for values 1 above their current maxes caused sketch to crash with a "Sketch not resolved" error after spending a long time running. I was unable to figure out why this happened. I initially thought that it could be because they are recursive sorting algorithms, unlike the other 3, however when I tried increasing the –bnd-inline-amnt command line option, which should have solved that error, it kept happening.

For selection insertion and bubble sort specifically, their running times all grew at roughly the same rate, with some minor differences that can likely be attributed to the different number of holes in each algorithm. In a few cases, such as with selection sort from n=4 to n=5, the running time actually decreased. Due to the fact that this only happened for low values of n, I believe that this is simply due to the somewhat random nature in which sketch generates code.

With merge sort, I ran into some issues that I did not have with the other sorting algorithms. Sometimes when generating code for merge sort, the outputted code would be a functioning merge sort program. However, other times, the outputted code was incorrect and would not actually sort an array. What's stranger is that this seemed to be consistent for the same values of n. For instance, when running merge sort with a value of n=4, sketch would always produce faulty code. However, when running merge sort with a value of n=6, merge sort would always produce correct code. I tried using the 2 sketch command line flags to generate code and a test harness in order to test the faulty merge sort code, however these always said that the generated code passed all tests. This leads me to believe that the test harness that sketch produces to test its generated programs is faulty in some way. Merge sort sometimes producing faulty code would explain why the running times were so low for many values of n for merge sort. However, merge sort was still faster than the other sorting algorithms when it did produce correct code. Running merge sort with a size of n=16, the largest value I was able to use with merge sort, produced correct code in a lower time than all the other sorting algorithms took to produce code with a lower value of n.

The last observation I have about these results is the way in which sketch was sometimes able to generate code that fulfilled the assertions yet did it in a way that was unexpected. For instance, when running selection sort with a size of n=2, sketch produced a selection sort that, because of how it arrived at its solution, was able to sort an array of size n=2, but not any larger

```
    i = i - 30;
    __sa2 = i == (n - 0);
```

In this example, we are looking at the end of the inner loop of the outputted code. Normally, this is where selection sort would increment the index i by 1, and then check that it is still less than the array size n. However, because sketch was running with a size of n=2, it found that it could simply change i from its starting value of 0 to anything else, in this case -30, which would cause the loop to end.

If you are only sorting an array of size 2, this works, as it swaps the only 2 values in the array and then ends, but its not the behavior you would expect from selection sort. This sort of weird output with selection sort ended after raising the size of n, as then sketch was forced to find a more general-purpose solution to fill the holes. Nevertheless, I found it very interesting how sketch would simply find the first values that worked, and then use those, even if they aren't what the programmer intended.

Another instance of sketch producing unexpected code sometimes happened in the code it outputted for quick sort.

```
if(n == 0)
{
  _out = in_0;
  return;
}
```

Normally at the beginning of quicksort, you would check if the size of the current array is 1, and if it is, you would return that array so that the array can be sorted together at a higher level. In this case, I would expect the first line in this example code to be if(n == 1). However, sketch was able to figure out that if an array of size 1 went through the entire generated merge sort function, it wouldn't attempt to recursively call quick sort again, and cause errors, but would instead end up simply returning the original array. In a way, sketch was able to outsmart me by causing certain pieces of code to never be run if they weren't needed for the output to be correct, which I find very interesting.

References

The Sketch Programmers Manual: https://people.csail.mit.edu/asolar/manual.pdf