Lab 13

Published

Tools Needed: Hydra machine

[Step 1]

1. Download **sort.o**

2. Create a new assembly file called sort.S

3. The only file you will be editing is sort.S

4. For ALL sorting algorithms, you will sort integers in ASCENDING order, meaning -1, 0, 1, 2, 3, 4, 5, ...

5. You are free to create your own testing files. We will be testing with different sized data sets, so your code must handle them all!

6. Test your code by compiling "aarch64-linux-gnu-gcc -o sort sort.o sort.S" and then typing "./sort". Lists are randomly generated each time, so it is probably best you try running the test more than once.

7. RESTRICTIONS

a) You must write these algorithms in ARMv8 Assembly

b) You may not call any functions within your assembly functions except:

- In QuickSort, you may call other QuickSort functions
- In GreatestProduct, you may call any one sorting function

c) You may not use any globals. The only section you'll use in this lab is .text

d) DO NOT pass off an algorithm for what it is not intended for. For example, if you put bubble sort for selection and/or insertion, you will not get credit for either. **This is to keep you honest when the TA grades your work.**

[Step 2: Bubble Sort]

1. Write the assembly instructions for BubbleSort.

2. You may not use any globals.

3. Your function (if it was written in C++) would be prototyped as

```
void BubbleSort(int *values, int size);
```

4. *values* contains *size* number of integers. You must use this array to sort in-place, meaning you cannot declare another array to sort into.

5. Bubble sort is a sorting algorithm that sorts by repeatedly swapping adjacent elements that are out of order from the start of the array until the end. This process is repeated until no swaps are made.

[Step 3: Insertion Sort]

1. Write the assembly instructions for InsertionSort using the insertion sort algorithm.

2. Just like bubble sort, you must sort in-place and may NOT use any globals.

3. Insertion sort is prototyped as

```
void InsertionSort(int *values, int size);
```

[Step 4: Selection Sort]

1. Write the assembly instructions for SelectionSort using the selection sort algorithm.

2. Just like insertion and selection sort, you must sort in-place, and use NO globals.

3. Selection sort is prototyped as

```
void SelectionSort(int *values, int size);
```

[Step 5: String Length Encoded]

1. Write the instructions for StringLengthEncoded. However, every single instruction must be specified in its encoded, hexadecimal form. For example ret will be ".inst 0xd65f03c0".

2. The prototype for StringLengthEncoded is:

```
int StringLengthEncoded(const char *s);
```

3. This function will count the number of characters (NOT including the ending 0) in a c-style string.

4. There must be NO assembly instructions in this function. Instead, everything will be encoded by hand!

[Step 6: Quick Sort]

1. Write the assembly instructions for QuickSort using the quicksort algorithm below.

2. QuickSort is prototyped as

```
void QuickSort(int *values, int size);
```

3. You will need to create another function called "Partition" and make use of the stack!

4. Just like every other sorting function, you must sort in-place and use no globals!

See below for how QuickSort is implemented in C++.

[Quicksort Algorithm]

Quicksort has three functions, but it is called QuickSort(elements, size) from sort.o (the driver code).

You will notice that QuickSortRun actually calls QuickSortRun twice. This is what is known as a "recursive" function, which you will learn more about in COSC140:

```
void QuickSort(int *elements, int size)
{
     QuickSortRun(elements, 0, size-1);
}

int QuickSortPartition(int *elements, int lo, int hi)
{
     int pivot;

     pivot = elements[lo];
     lo = lo - 1;
     hi = hi + 1;

     do {
          do {
               lo = lo + 1;
          } while (elements[lo] < pivot);

          do {
               hi = hi - 1;
          } while (elements[hi] > pivot);

          if (lo >= hi) {
               break;
          }
          swap(elements[lo], elements[hi]);
     } while (true);

     return hi;
}

void QuickSortRun(int *elements, int lo, int hi)
{
     int part;
     if (lo < hi) {
```

```
        part = QuickSortPartition(elements, lo, hi);
        QuickSortRun(elements, lo, part);
        QuickSortRun(elements, part+1, hi);
    }
}
```

[Step 7: GreatestProduct]

1. Write the assembly instructions for GreatestProduct.

2. GreatestProduct is prototyped as

```
int GreatestProduct(int *values, int num_values);
```

3. The only operation you have available to you is that you can only increment only one of the values by one. The range of each individual value is 1..50.

4. The purpose of this function is to find the greatest product value of values[0] * values[1] * ... * values[num_values-1] (the product of all values with only one of them being incremented by 1).

5. The naive approach is to try all combinations (values[0]+1) * values[1] * ... * values[num_values-1], values[0]*(values[1]+1) * ... * values[num_values-1], but this is extremely slow and not efficient.

6. Your job is to figure out how to solve this problem in only ONE try.

7. GreatestProduct returns the greatest possible product of all of the values[] with one (and only one) of them being incremented by 1.

HINT: you may call one of your sorting algorithms.

[Step 8: BinarySearch]

1. Write the assembly instructions for the function BinarySearch

2. The prototype of this function is:

```
int BinarySearch(const int *haystack, int size, int needle);
```

3. This function will search through the haystack array looking for the value given by needle. It will then return the INDEX of the value that is needle. If needle was not found, return -1.

4. You must use the binary searching algorithm for this function! Your function will be given a sorted list.

[You are finished with this lab!]

Submit your lab by uploading sort.S as sort.txt, including BinarySearch, GreatestProduct, StringLengthEncoded, BubbleSort, InsertionSort, SelectionSort, and QuickSort into the provided text box.

Points	200
Submitting	a file upload
File Types	txt

Due	For	Available from	Until
Nov 30, 2017	Everyone else	Nov 17, 2017 at 7:58am	Nov 30, 2017 at 11:59pm
Dec 6, 2017	1 student	-	Dec 6, 2017 at 11:59pm

Lab 13

Criteria	Ratings					Pts
BubbleSort -No assembly globals allowed!	**25.0 pts** **Full Marks**	**15.0 pts** **Minor Problems**	**10.0 pts** **Some Problems**	**5.0 pts** **Major Problems**	**0.0 pts** **No Marks**	25.0 pts
InsertionSort -No assembly functions allowed!	**30.0 pts** **Full Marks**	**24.0 pts** **Minor Problems**	**17.0 pts** **Some Problems**	**10.0 pts** **Major Problems**	**0.0 pts** **No Marks**	30.0 pts
SelectionSort -No assembly globals allowed!	**30.0 pts** **Full Marks**	**24.0 pts** **Minor Problems**	**17.0 pts** **Some Problems**	**10.0 pts** **Major Problems**	**0.0 pts** **No Marks**	30.0 pts
QuickSort -No assembly globals allowed!	**40.0 pts** **Full Marks**	**34.0 pts** **Minor Problems**	**28.0 pts** **Some Problems**	**11.0 pts** **Major Problems**	**0.0 pts** **No Marks**	40.0 pts
StringLengthEncoded -No assembly globals allowed! -All instructions must be encoded using .inst and either a binary or a hexadecimal number	**20.0 pts** **Full Marks**	**15.0 pts** **Minor Problems**	**10.0 pts** **Some Problems**	**5.0 pts** **Major Problems**	**0.0 pts** **No Marks**	20.0 pts
GreatestProduct -Calls a sorting function -Properly increments the appropriate element (only one!) -Returns the correct product	**40.0 pts** **Full Marks**	**34.0 pts** **Minor Problems**	**28.0 pts** **Some Problems**	**9.0 pts** **Major Problems**	**0.0 pts** **No Marks**	40.0 pts
BinarySearch	**35.0 pts** **Full Marks**	**29.0 pts** **Minor Problems**	**21.0 pts** **Some Problems**	**13.0 pts** **Major Problems**	**0.0 pts** **No Marks**	35.0 pts
						Total Points: 220.0