CS3340.003 Final Project Report
Various Sorts

Professor: Dr. Karen Mazidi
Team Members: Naman Garg and Kaushik Nadimpalli

Project Description

The purpose of this project is to implement various sorting algorithms that are commonly used in higher-end programming languages in MIPS. The program seeks to sort a list/set of n positive numbers in increasing order. In order to achieve this, we will be using three different sorting algorithms and demonstrating their sorting mechanisms in MIPS. The three sorting algorithms we implemented are: Heap Sort, Selection Sort, and Bubble Sort. The user will be asked to enter the number of elements he/she wishes to be sorted. From there, they enter those values and choose between three options. Option 1 performs Heap Sort on the list of numbers. Option 2 does the same but instead executes bubble sort. Option 3 executes Selection Sort. The main goal of implementing these 3 specific algorithms is to determine the number of instructions executed to sort the same list. Each of the 3 sorts is inherently more efficient in sorting a specific type of list. Different test cases will give different results. More information about test cases will be addressed in another section of the report. Overall, our goal is to demonstrate the implementation of these algorithms and to distinguish their ability to sort lists in terms of efficiency and performance levels.

Updated Code(Note to Professor and TA)

If an option other than 1, 2 , or 3 is selected, then the program automatically sorts the list using all three algorithms. This way, if the user wishes to compare how many instructions it takes to sort the data using more than one of these algorithms, they just have to enter the data once rather than building and rerunning the program. This feature saves time on the user's end, enabling data to be entered only once.

Sorting - Advantages and Usefulness in MIPS

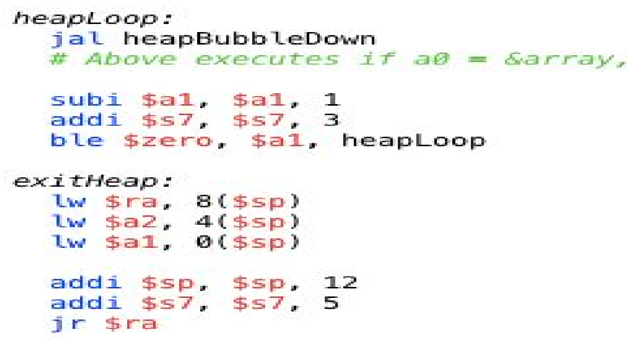
Sorting, in its basic definition, is the arrangement of an unordered list into that of an ordered list. While there are no exact standards on how list is sorted, they usually follow two principles: the order must be in non-decreasing order and the output must be a permutation of the input. Compared to the implementation of these algorithms in higher-level languages such as Java and C++, MIPS gives us a more accurate count of how many instructions were executed during each sort. This gives us a clear idea and an incentive to be more decisive when it comes to what functions we are using in the implementation of these algorithms.

Program Structure - Converting a String of Numbers to an Array of Numbers

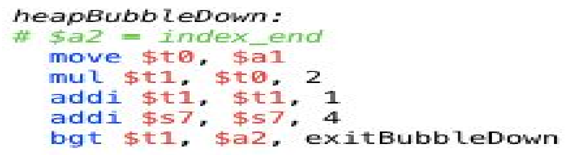
In this program, the user will be asked to enter the how many positive numbers they wish to sort which will be stored. Then the user will enter the list of numbers separated by spaces which will be stored as a string. From there, we use a parser function to convert the string of numbers into an array of integers. This is accomplished by reading in the ascii value and subtracting 48. The function also reads spaces to determine the start/end of a new number. Once the array is formed, it will be used in all the other sorting algorithms. We also have an array copy function that is useful if we are doing all 3 sorts at once (since we don't want to modify and lose the original list).

Then the user is prompted to choose which sorting algorithm to perform by selecting (1- Heap, 2-Bubble, 3-Selection). If the user enters any other value, all 3 sorts are automatically performed.

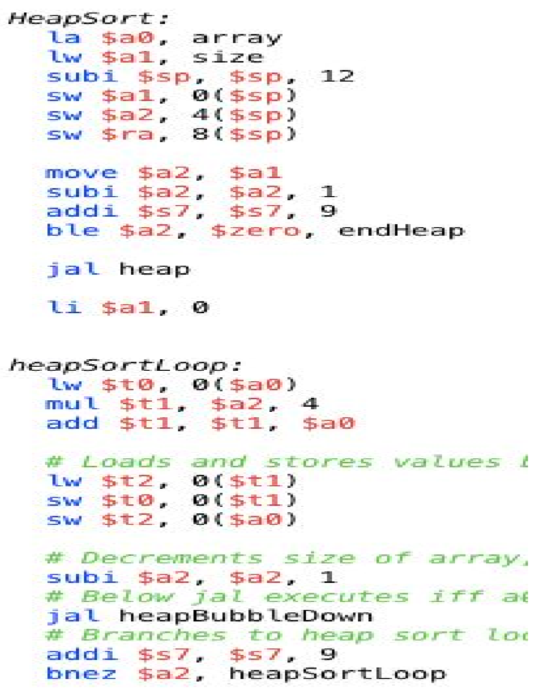
Heap Sort



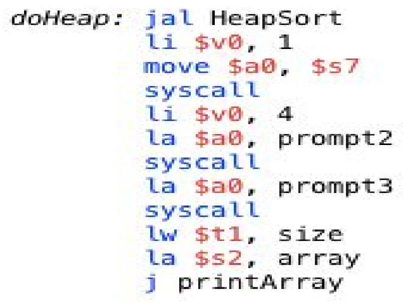
```
heapLoop:
   jal heapBubbleDown
   # Above executes if a0 = &array,

   subi $a1, $a1, 1
   addi $s7, $s7, 3
   ble $zero, $a1, heapLoop

exitHeap:
   lw $ra, 8($sp)
   lw $a2, 4($sp)
   lw $a1, 0($sp)

   addi $sp, $sp, 12
   addi $s7, $s7, 5
   jr $ra
```



```
heapBubbleDown:
# $a2 = index_end
   move $t0, $a1
   mul $t1, $t0, 2
   addi $t1, $t1, 1
   addi $s7, $s7, 4
   bgt $t1, $a2, exitBubbleDown
```



```
HeapSort:
   la $a0, array
   lw $a1, size
   subi $sp, $sp, 12
   sw $a1, 0($sp)
   sw $a2, 4($sp)
   sw $ra, 8($sp)

   move $a2, $a1
   subi $a2, $a2, 1
   addi $s7, $s7, 9
   ble $a2, $zero, endHeap

   jal heap

   li $a1, 0

heapSortLoop:
   lw $t0, 0($a0)
   mul $t1, $a2, 4
   add $t1, $t1, $a0

   # Loads and stores values i
   lw $t2, 0($t1)
   sw $t0, 0($t1)
   sw $t2, 0($a0)

   # Decrements size of array,
   subi $a2, $a2, 1
   # Below jal executes iff a
   jal heapBubbleDown
   # Branches to heap sort lo
   addi $s7, $s7, 9
   bnez $a2, heapSortLoop
```



```
doHeap: jal HeapSort
        li $v0, 1
        move $a0, $s7
        syscall
        li $v0, 4
        la $a0, prompt2
        syscall
        la $a0, prompt3
        syscall
        lw $t1, size
        la $s2, array
        j printArray
```

<u>Overview - Heap Sort MIPS Implementation</u>

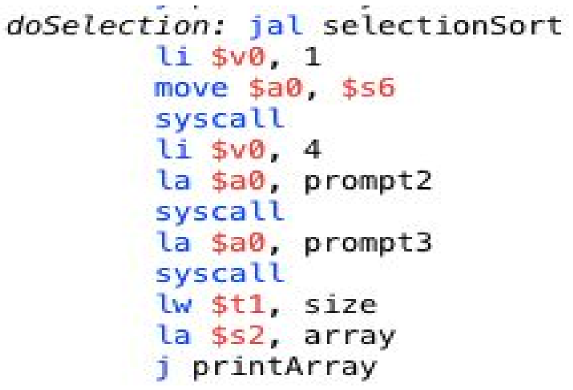
1) Build a max heap from the input data. A max heap is a binary tree in which the parent node is always larger than its children
2) At this point, the largest item is stored at the root of the heap.
3) Find the last children that do not exceed the end of the list. In a list, the children of a node at position n are located at positions 2n and 2n+1.
4) Determine which child is smaller (if there are 2 children)
5) Swap the root and the smaller child followed by removing the last item of the heap (which will be the largest element after the swap)
6) Heapify the root of the tree so that it is once again a max heap.
7) Repeat above steps while size of heap is greater than 1.

Overall time complexity is O(nlogn).

This is because building the max heap is O(n) and heapifying it removing the last node is O(logn).

CS3340.003 Final Project Report
Various Sorts
Professor: Dr. Karen Mazidi
Team Members: Naman Garg and Kaushik Nadimpalli

Selection Sort



```
doSelection: jal selectionSort
        li $v0, 1
        move $a0, $s6
        syscall
        li $v0, 4
        la $a0, prompt2
        syscall
        la $a0, prompt3
        syscall
        lw $t1, size
        la $s2, array
        j printArray
```

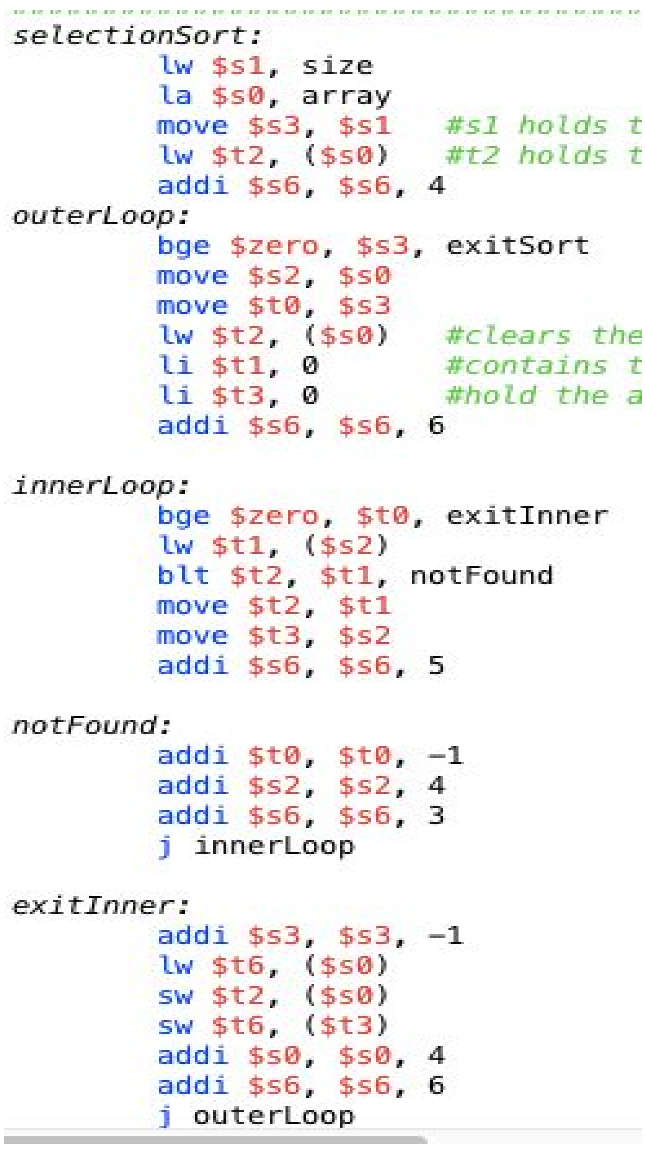
Overview - Selection Sort MIPS Implementation

1) Traverse the array using a counter.
2) Store the first element in a separate register.
3) Compare all the registers to the register containing the smallest element.
4) If another element is smaller, store the smaller element and its address.
5) Repeat this process until you reach the end of the array
5) Swap the smallest element with the first element.

3) Find the smallest element again but when searching through the array, start at the 2nd element (since we already know the first element is the smallest)
4) Once the whole has been traversed, swap the 2nd smallest element with the 2nd element.
5) Continue to do this until you reach the end of the list.

Overall the time complexity is O(n^2).

Selection sort is a relatively inefficient algorithm. Although it performs better for smaller lists than other sorting algorithms that have a O(nlogn), it is still inferior to bubble since bubble is adaptive and can stop traversing the list once the list is in order. With selection however, the sort only stops when the whole list has been traversed.



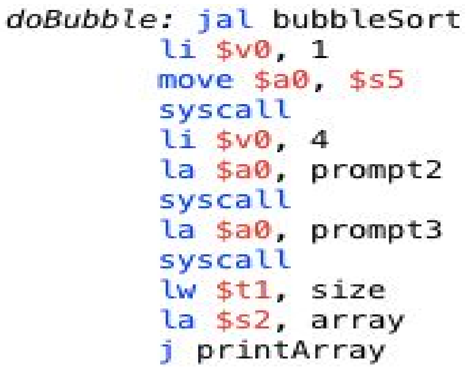
```
selectionSort:
        lw $s1, size
        la $s0, array
        move $s3, $s1    #s1 holds t
        lw $t2, ($s0)    #t2 holds t
        addi $s6, $s6, 4
outerLoop:
        bge $zero, $s3, exitSort
        move $s2, $s0
        move $t0, $s3
        lw $t2, ($s0)    #clears the
        li $t1, 0        #contains t
        li $t3, 0        #hold the a
        addi $s6, $s6, 6

innerLoop:
        bge $zero, $t0, exitInner
        lw $t1, ($s2)
        blt $t2, $t1, notFound
        move $t2, $t1
        move $t3, $s2
        addi $s6, $s6, 5

notFound:
        addi $t0, $t0, -1
        addi $s2, $s2, 4
        addi $s6, $s6, 3
        j innerLoop

exitInner:
        addi $s3, $s3, -1
        lw $t6, ($s0)
        sw $t2, ($s0)
        sw $t6, ($t3)
        addi $s0, $s0, 4
        addi $s6, $s6, 6
        j outerLoop
```

CS3340.003 Final Project Report
Various Sorts

Professor: Dr. Karen Mazidi

Team Members: Naman Garg and Kaushik Nadimpalli

Bubble Sort



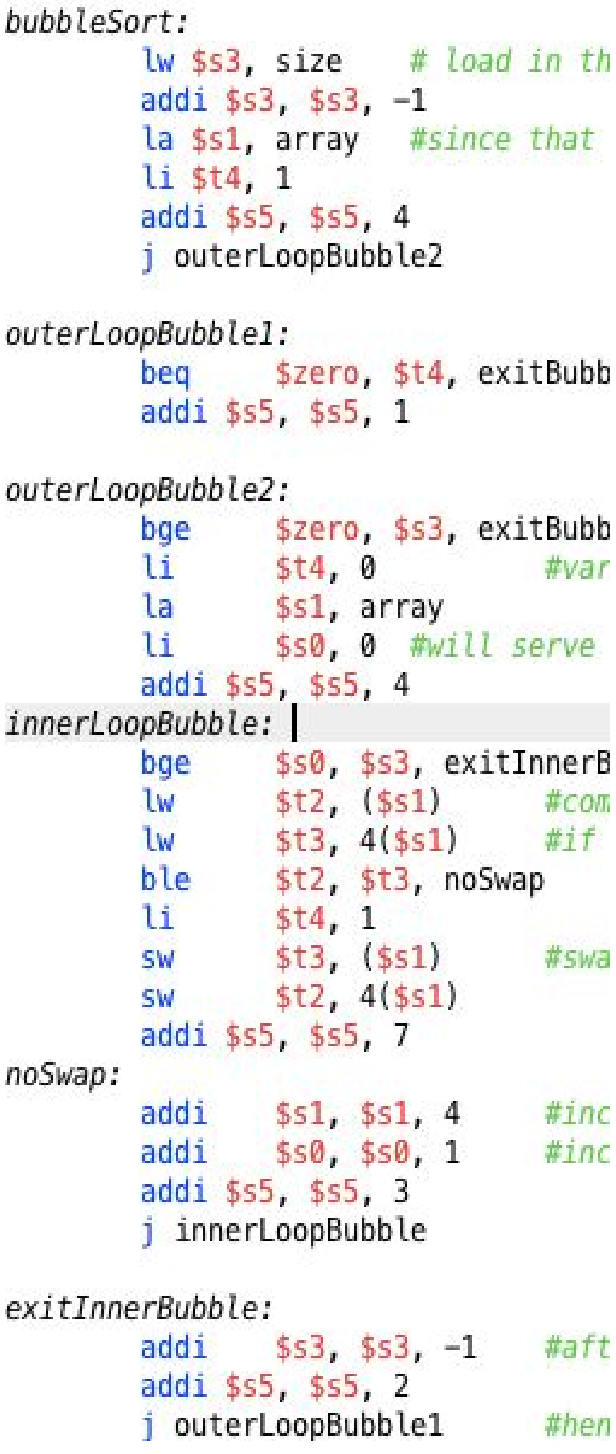
```
doBubble: jal bubbleSort
          li $v0, 1
          move $a0, $s5
          syscall
          li $v0, 4
          la $a0, prompt2
          syscall
          la $a0, prompt3
          syscall
          lw $t1, size
          la $s2, array
          j printArray
```

Overview - Bubble Sort

Similar to selection, bubble sort seeks to repeatedly move the largest element in the list to the highest index position of the array. As in selection sort, each iteration reduces the effective size of the array (we already sorted it once and placed the largest element in its rightful position). The two algorithms differ in how this is done. Rather than search the “entire effective array to find the largest element, bubble sort focuses on successive adjacent pairs of elements in the array, compares them, and either swaps them or moves on to the next pair.” In either case, after this step, the larger of the two elements will be in the higher index position.

Overall the time complexity is O(n^2).

However, bubble sort is adaptive and can end early without having to traverse the entire list if the list is sorted. This is accomplished if during an iteration no swap has been made. We keep track of this by looking at register $t4. 0 indicates no swap was made and 1 indicates a swap was made.



```
bubbleSort:
        lw $s3, size    # load in th
        addi $s3, $s3, -1
        la $s1, array   #since that
        li $t4, 1
        addi $s5, $s5, 4
        j outerLoopBubble2

outerLoopBubble1:
        beq     $zero, $t4, exitBubb
        addi $s5, $s5, 1

outerLoopBubble2:
        bge     $zero, $s3, exitBubb
        li      $t4, 0          #var
        la      $s1, array
        li      $s0, 0  #will serve
        addi $s5, $s5, 4
innerLoopBubble:
        bge     $s0, $s3, exitInnerB
        lw      $t2, ($s1)      #com
        lw      $t3, 4($s1)     #if
        ble     $t2, $t3, noSwap
        li      $t4, 1
        sw      $t3, ($s1)      #swa
        sw      $t2, 4($s1)
        addi $s5, $s5, 7
noSwap:
        addi    $s1, $s1, 4     #inc
        addi    $s0, $s0, 1     #inc
        addi $s5, $s5, 3
        j innerLoopBubble

exitInnerBubble:
        addi    $s3, $s3, -1    #aft
        addi $s5, $s5, 2
        j outerLoopBubble1      #hen
```

For each algorithm, we keep track of the number of instructions executed. For example, if seven instructions are executed before a branch statement, then we will increase the instruction

counter by 8. Once the sorting has completed, the array will be displayed using the printArray function along with approximately how many instructions were executed.

Assumptions and Testing(How we tested the program)

1) The user is expected to only enter positive integers separated by spaces
2) The program does not any other characters except numbers and spaces
3) The user is required to provide the correct number of elements in the list that he/she wants to sort
    a) For example, say the user wants to sort a list of 20 positive integers.
        i) If the user enters less than 20 values, the program will fill the list with 0s to reach 20 values.
        ii) If the user enters more than 20 values, the program will only read the first 20 values.
4) **Updated Code Assumption:** In the case the user did not pick Options 1-3, and instead picked another Option(4-9 and 0), the program will run all three algorithms and print the sorted lists and the number of instructions it took for that list to be sorted under each algorithm.

We tested the code with various list sizes with various levels of "randomness". There are small lists, medium list, large lists, sorted lists, partially sorted lists, random lists.

Test Cases Samples

Short/Random: 60 5 54 44 100

Medium/Random: 25 16 73 1 32 42 62 96 21 68 57 43 6 1 11 49 24 95 7 77

Large/Random: 25 16 73 1 32 42 62 96 21 68 57 43 6 1 11 49 24 95 7 77 25 16 73 1 32 42 62 96 21 68 57 43 6 1 11 49 24 95 7 77

Sorted: 1 2 3 4 5

Partially Sorted: 1 2 3 4 5 60 5 54 44 100

Observations

We observe that for short lists, selection and bubble execute far less instructions than heap. However, as we sort larger lists, the heap sort executes far less instructions than bubble or heap. For example, for the short test case, bubble was done in under 100 instructions and selection was done in under 150 instructions. Heap however took over 200 instructions. For the large test case, bubble executed over 5000 instructions while heap only executed 3000 instructions. Since bubble is adaptive, we noticed that for the sorted list it executed very few instructions (22) in comparison to selection (134) and heap (205). This makes sense and further accentuates the efficiency of bubble sort on partially or fully sorted lists.

CS3340.003 Final Project Report
Various Sorts
Professor: Dr. Karen Mazidi
Team Members: Naman Garg and Kaushik Nadimpalli

Significance of project

The underlying purpose of this project is to implement some of the most basic algorithms used in programming: sorts. MIPS, being a RISC instruction set architecture, enables us to see how basic operations, actions such as printing, and functions work at a much deeper level in comparison to the paradigm using in higher level languages such as C, C++, or Java. We can see the way information is being stored in memory and how we are pulling/accessing that information much more vividly in an assembly language. Basic sorting algorithms such as Bubble and Selection as well as more advanced ones can be implemented in higher level languages quite easily. The same could not be said in an assembly language such as MIPS, due to the mere fact that we have to keep track of hundreds of lines containing functions, branch jumps, loops, as well as gathering information from or storing information to arrays. We wanted to showcase how various sorting algorithms work in MIPS, and how our knowledge of their behavior from languages such as C or Java can be augmented by seeing their structural implementation in MIPS. As seen in the project itself, we added step by step comments describing what each line of code is doing, and explained how it contributes to development of the overall algorithm it is a part of. Understanding and relating the functionality of these algorithms such as these help us visualize how the CPU and hardware elements of the computer interact/build on each other to construct these sorts.

Furthermore, this program can be useful in other programs in which you are expecting to sort some list of numbers. Depending on the type of list (big/small, random/partially sorted), we can choose which sorting algorithm to implement by comparing how many instructions are executed. This will allow for more efficiency and faster sorting.

Sources

“5.7. The Bubble Sort¶.” ***5.7. The Bubble Sort — Problem Solving with Algorithms and Data Structures***, Interactive Python, interactivepython.org/runestone/static/pythonds/SortSearch/TheBubbleSort.html.

“Selection Sort Algorithm.” ***Online CS Modules: The Selection Sort***, Virginia Tech CS Department, courses.cs.vt.edu/csonline/Algorithms/Lessons/SelectionSort/index.html.

BC, Aarya. “AaryaBC/HeapSort-Mips.” ***GitHub***, Github, 20ADAD.

“Heap Sort.” ***GeeksforGeeks***, 9 Oct. 2017, www.geeksforgeeks.org/heap-sort/.