Unique Array Sorting: Sorting with a twist!

Changes and Adjustments from phase 2:

I adjusted my code to have a set array, and use that array to step through each index, and push each digit onto the stack by utilizing push and pop commands. I changed my numbers to be like the array I used in phase 1, and made it hold 10 values like I had in phase 1 instead of the 6 values like I utilized in phase 2. Instead of manually storing the numbers into registers and pushing them onto the stack one by one, I made a loop that would run through each index of the array using next_element and pushed it onto the stack using pushDigit. I then shortened is_last and is_less by making a new function called print which would do shadow spacing, and stack alignment, and take the stored variable and print it out to the screen. I also added a variable that makes the code run 1500 times so that I could visually see the time.

Benchmark of the Original Program:

I am using my personal desktop, which contains an Intel i7 7700K @ 4.20GHz (team blue!) with no overclocking or underclocking since its 15 year old CPU and still running strong.

I am benchmarking my whole code since which includes all my function calls. After running my program on the unoptimized phase 3 code three times, I ended up with the following results:

462ms 476ms 471ms Average: 469.667ms

```
24      mainCRTStartup PROC
25          mov r14, 1500
26
27      start:
28          cmp r14, 0
29          jz end_code
30          mov r13, 0
31          lea r13, [array]
32          mov r11, 0
33          mov r12, 0
34          mov rbx, 0
35          mov r15, 0
36
37      next_element:
38          lea r10, [r13]
39          cmp byte ptr [r10], 0
40          je myloop
41          movzx r10, byte ptr [r13]
42          sub r10, '0'
43          imul r11, r11, 10
44          mov r11, r10
45          test r13, 1
46          jz pushDigit
47          inc r13
48          cmp r11, '0'
49          jl next_element
```

```
50
51      pushDigit:
52          push r11
53          inc r13
54          jmp next_element
55
56      myloop:
57          pop rsi
58          pop rdi
59          call check_index
60          cmp rax, 1
61          je odd_number
62
63      even_number:
64          cmp r12, 8
65          je is_less
66          cmp rdi, rsi
67          jg print
68
69      odd_number:
70          cmp rdi, rsi
71          jl is_less
72          test rbx, 1
73          jnz is_less
74
75
76      is_greater:
77          cmp rbx, 8
78          push rsi
79          push rdi
80          pop rdi
81          pop rsi
82          je is_less
83
84      last:
85          mov rcx, offset myString
86          mov rdx, rsi
87          jmp print
88
89
90      is_less:
91          push rsi
92          push rdi
93          pop rsi
94          pop rdi
95          mov rcx, offset myString
96          mov rdx, rsi
97          jmp print
```

```
98
99      print:
100         sub rsp, 32
101         inc r12
102         call printf
103         add rsp, 32
104         push rdi
105         inc rbx
106         cmp rbx, 9
107         jge stop
108         jmp myloop
109
110     stop:
111         mov rcx, offset myString
112         mov rdx, rdi
113         call printf
114         dec r14
115         jmp start
116
117     end_code:
118         mov ecx, 0  ≤ 462ms elapsed
119         call ExitProcess
120     mainCRTStartup ENDP
121     END
```

Optimization #1 – Function Inlining and cheaper instructions

When I was looking for my first optimization, I noticed that some of my functions would cause my program to jump back when I could have simply merged some of my functions and order them so that they wouldn't have to jump back in the code as much. I started by getting rid of the 2 functions called even_number and odd_number. After removing the 4 conditions within those functions I then was able to make it a singular condition that would configure where to jump next and added to the end of the myloop function. I then removed the function called last since I noticed the function above it would just jump over that function and deleted the jmp statement in is_greater since it would always go to is_less, I simply moved is_less below is_greater so that it would be inline. Once I got all of that completed, I was then able to get rid of my print function. Instead of having that function I was able to move it into the function called is_less and make it have cheaper instructions and not have to run as many lines of code. I also noticed that as I would run through the code, I would naturally swap rsi and rdi by pushing them onto the stack and pulling them off, and in is_less I would do it again causing it to have extra instruction and run time, so to fix this I got rid of the swapping in is_less, and made a condition to jump over is_greater so it wouldn't swap in the first place. . After running my program on the optimized phase 3 code three times, I ended up with the following results:

421ms **418ms** **427ms** **Average: 422ms** **Speed Increase: ~10%**

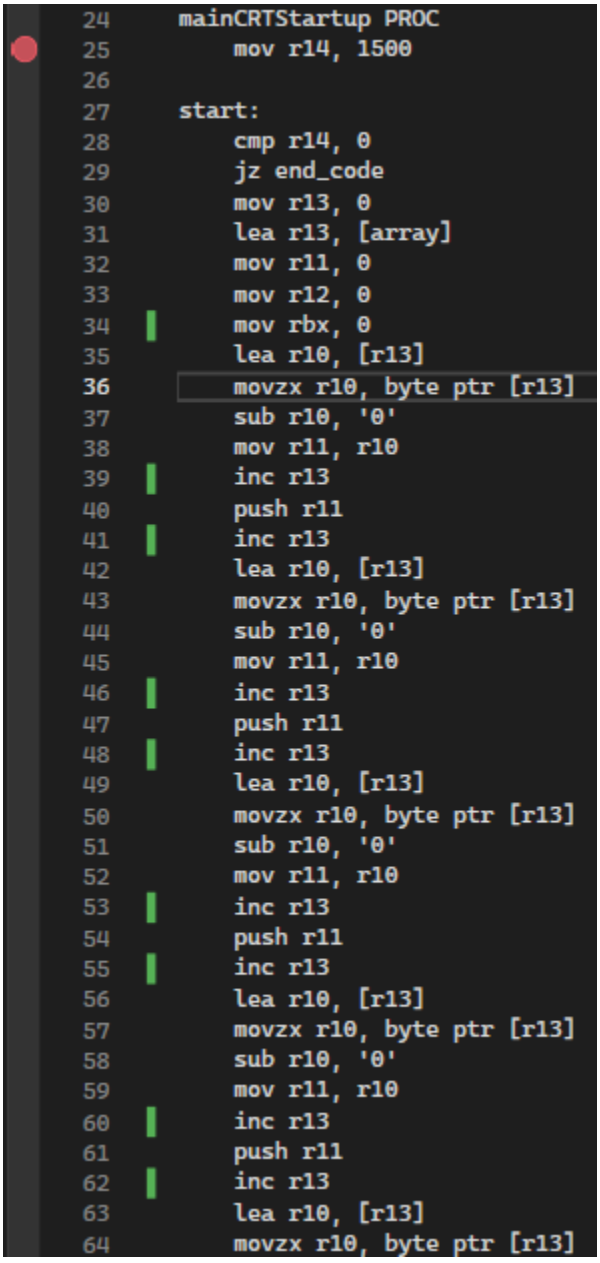
```
24      mainCRTStartup PROC
25          mov r14, 1500
26
27      start:
28          cmp r14, 0
29          jz end_code
30          mov r13, 0
31          lea r13, [array]
32          mov r11, 0
33          mov r12, 0
34          mov rbx, 0
35
36      next_element:
37          lea r10, [r13]
38          cmp byte ptr [r10], 0
39          je myloop
40          movzx r10, byte ptr [r13]
41          sub r10, '0'
42          imul r11, r11, 10
43          mov r11, r10
44          test r13, 1
45          jz pushDigit
46          inc r13
47          cmp r11, '0'
48          jl next_element
49
50      pushDigit:
51          push r11
52          inc r13
53          jmp next_element
```

```
54
55      myloop:
56          pop rsi
57          pop rdi
58          call check_index
59          cmp rax, 1
60          je is_greater
61          cmp rsi, rdi
62          jl is_less
63
64      is_greater:
65          cmp rbx, 8
66          push rsi
67          push rdi
68          pop rsi
69          pop rdi
70
71      is_less:
72          mov rcx, offset myString
73          mov rdx, rsi
74          sub rsp, 32
75          sub rsp, 8
76          call printf
77          add rsp, 40
78          push rdi
79          inc rbx
80          cmp rbx, 9
81          jge stop
82          jmp myloop
83
84      stop:
85          mov rcx, offset myString
86          mov rdx, rdi
87          call printf
88          dec r14
89          jmp start
90
91      end_code:
92          mov ecx, 0  ≤ 421ms elapsed
93          call ExitProcess
94      mainCRTStartup ENDP
95      END
```

Optimization #2 – Loop Unrolling

In my code, I have a loop that is used quite often at the beginning of my code, and it is used to push my array onto the stack. I stripped the loop down and made it to where it would use the minimum number of lines. As I was running through it I realized I was actually able to get rid of a decent amount of code. As you can tell in my original code, the function next_element has 12 lines of code, and once I unrolled the loop I ran through each line and watched the values and played around commenting out useless lines of code to see what they effected, and I was able to shrink it down to 5 lines of code. After unrolling the loop, I ran my optimized program three times, resulting in these speeds:

429ms **422ms** **423ms** **Average: 424.667ms** **Speed Increase: ~10%**



```
24  mainCRTStartup PROC
25      mov r14, 1500
26
27  start:
28      cmp r14, 0
29      jz end_code
30      mov r13, 0
31      lea r13, [array]
32      mov r11, 0
33      mov r12, 0
34      mov rbx, 0
35      lea r10, [r13]
36      movzx r10, byte ptr [r13]
37      sub r10, '0'
38      mov r11, r10
39      inc r13
40      push r11
41      inc r13
42      lea r10, [r13]
43      movzx r10, byte ptr [r13]
44      sub r10, '0'
45      mov r11, r10
46      inc r13
47      push r11
48      inc r13
49      lea r10, [r13]
50      movzx r10, byte ptr [r13]
51      sub r10, '0'
52      mov r11, r10
53      inc r13
54      push r11
55      inc r13
56      lea r10, [r13]
57      movzx r10, byte ptr [r13]
58      sub r10, '0'
59      mov r11, r10
60      inc r13
61      push r11
62      inc r13
63      lea r10, [r13]
64      movzx r10, byte ptr [r13]
```

```
        sub r10, '0'
        mov r11, r10
        inc r13
        push r11
        inc r13
        lea r10, [r13]
        movzx r10, byte ptr [r13]
        sub r10, '0'
        mov r11, r10
        inc r13
        push r11
        inc r13
        lea r10, [r13]
        movzx r10, byte ptr [r13]
        sub r10, '0'
        mov r11, r10
        inc r13
        push r11
        inc r13
        lea r10, [r13]
        movzx r10, byte ptr [r13]
        sub r10, '0'
        mov r11, r10
        inc r13
        push r11
        inc r13
        lea r10, [r13]
        movzx r10, byte ptr [r13]
        sub r10, '0'
        mov r11, r10
        inc r13
        push r11
        inc r13
        lea r10, [r13]
        movzx r10, byte ptr [r13]
        sub r10, '0'
        mov r11, r10
        inc r13
        push r11
        inc r13

    myloop:
        pop rsi
        pop rdi
        call check_index
        cmp rax, 1
        je odd_number

    even_number:
        cmp r12, 8
        je is_less
        cmp rdi, rsi
        jg print

    odd_number:
        cmp rdi, rsi
        jl is_less
        test rbx, 1
        jnz is_less


    is_greater:
        cmp rbx, 8
        push rsi
        push rdi
        pop rdi
        pop rsi
        je is_less

    last:
        mov rcx, offset myString
        mov rdx, rsi
        jmp print


    is_less:
        push rsi
        push rdi
        pop rsi
        pop rdi
        mov rcx, offset myString
        mov rdx, rsi
        jmp print

    print:
        sub rsp, 32
        inc r12
        call printf
        add rsp, 32
        push rdi
        inc rbx
        cmp rbx, 9
        jge stop
        jmp myloop

    stop:
        mov rcx, offset myString
        mov rdx, rdi
        call printf
        dec r14
        jmp start

    end_code:
        mov ecx, 0  ≤ 429ms elapsed
        call ExitProcess
    mainCRTStartup ENDP
    END
```