Question 5 (30 points):

Null-terminated strings are sorted based on a lexicographic comparison of their characters. Given two strings S_a and S_b, the result of their lexicographic comparison can have these outcomes:

- $S_a = S_b$ if $S_a[i] = S_b[i]$ for every character in S_a and in S_b.
- $S_a < S_b$ if $\exists j$ such that $S_a[i] = S_b[i]$ for every $i < j$ and $S_a[j] < S_b[j]$.

Therefore, to lexicographically compare two strings, we compare the ASCII codes of the strings character by character until we find the first position in which the strings are different. Whichever string has a smaller character at that position is the smaller string.

When comparing two strings of different lengths, if one of the strings is a substring of the other, the shorter string is lexicographically smaller. For example the null-terminated string `grade` is lexicographically smaller than the null-terminated string `gradeA`.

In this question you will create two functions to sort a vector of pointers to strings. Lets call this vector `S`. Each element of `S` contains a pointer to a null-terminated string (the address of the first character of a null-terminated string). The end of `S` is signalled by the sentinel value -1.

The task is to sort the pointers in `S` such that the strings are sorted in lexicographic order. To accomplish this task, you will write two functions in RISC-V assembly. The first function, `LexiCmp`, compares two string and returns -1, 0, or 1, depending on the result of the lexicographic comparison of the two strings. The second function, `LexiSort`, sorts the pointers in `S` according to their lexicographic order. `LexiSort` must call `LexiCmp` in order to compare strings.

Both functions must follow all the RISC-V register saving and restoring conventions. The two functions must be written independently, thus no assumptions about how the other function uses registers can be made.

The solution must work for any null-terminated strings, including the empty string. An empty string is lexicographically smaller than any non-empty string.

1. (**10 points**) Write RISC-V assembly code for a function called `LexiCmp`. It has two parameters that are pointers to strings and a single return value.

   parameters:
   - `a0`: pointer to a string S_a
   - `a1`: pointer to a string S_b

   return value:
   - `a0 = 0` if $S_a = S_b$
   - `a0 = 1` if $S_a > S_b$
   - `a0 = -1` if $S_a < S_b$

2. (**20 points**) Write `LexiSort`, which implements the Bubble Sort algorithm to sort the pointers to strings in `S`. The single parameter of `LexiSort` is the address of the first string pointer in `S`. `LexiSort` has no return values. After the execution of `LexiSort` the pointers in `S` are sorted according to the lexicographic order of the strings.

```
# LexiCmp
# arguments: a0: pointer to string Sa
#            a1: pointer to string Sb
# return value: a0: -1 if Sa<Sb; 1 if Sa>Sb; 0 if Sa=Sb
#
LexiCmp:
NextChar:
    lb      t0, 0(a0)           # t0 <- *Sa
    lb      t1, 0(a1)           # t1 <- *Sb
    blt     t0, t1, LessThan    # if t0<t1 then Sa<Sb
    bgt     t0, t1, GreaterThan # if t0>t1 then Sa>Sb
    addi    a0, a0, 1           # Sa++
    addi    a1, a1, 1           # Sb++
    bne     t0, zero, NextChar  # Not end of string
    mv      a0, zero            # Sa=Sb thus a0 <- 0
    jalr    zero, ra, 0
LessThan:
    addi    a0, zero -1         # a0 <- -1
    jalr    zero, ra, 0
GreaterThan:
    addi    a0, zero, 1         # a0 <- 1
    jalr    zero, ra, 0
```

Figure 1: A solution for `LexiCmp`.

```
void LexiSort(char **S){
    int i, j;
    int Ctemp;
    char *Stempj, *Stempj1;
    for (i = 0; S[i] != -1; i += 1){
        for (j = i-1; j >= 0; j -= 1){
            Stempj = S[j];
            Stempj1 = S[j+1];
            Ctemp = LexiCmp(Stempj,Stempj1);
            if(Ctemp <= 0){
              break;
            }
            S[j] = Stempj1;
            S[j+1] = Stempj;
        }
    }
}
```

(a) Index-based Sorting

```
void LexiSort(char **S){
    int Ctemp;
    char *Rtemp, *Rtemp1;
    char **T, **R;
    for (T = S ; *T != -1; T++){
        R = T;
        for (R--; R >= S; R--){
            Rtemp = *R;
            Rtemp1 = *(R+1)
            Ctemp = LexiCmp(Rtemp,Rtemp1);
            if(Ctemp <= 0){
              break;
            }
            *R = Rtemp1;
            *(R+1) = Rtemp;
        }
    }
}
```

(b) Pointer-based Sorting

Figure 2: C code for two alternative versions of Bubble Sorting.

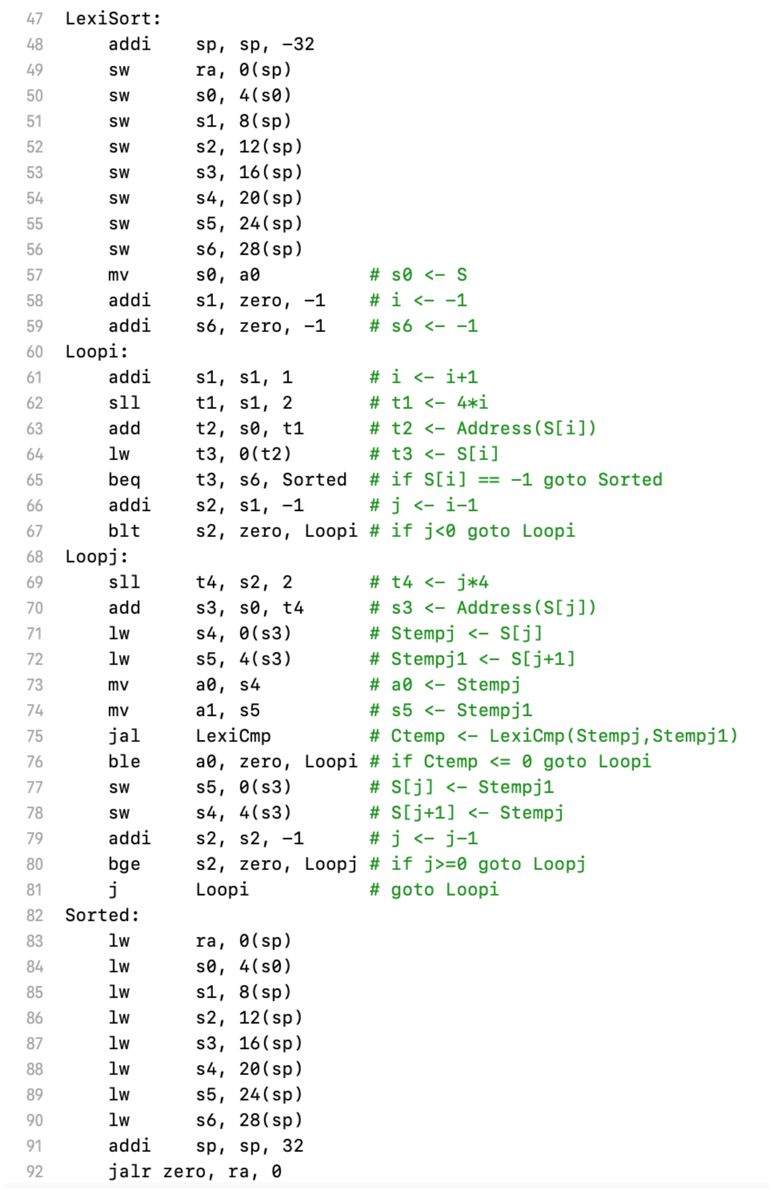
Figure 1 provides two alternative C versions for `LexiSort`. One version uses indexes to access the positions of the vector `S`, while the other version is based on pointers. You are free to use either of these versions as a basis for your RISC-V implementation of `LexiSort`. You may also create your own implementation of Bubble Sort as long as it correctly sorts the string pointers in the vector `S`. If using the pointer-based version, remember that the C compiler increments or decrements pointers based on their types. Thus, if a pointer `P` points to an item that is stored in `k` bytes, the C compiler will replace a statement such as `P++` by the statement `P=P+k`. Both versions of the C code above use a `break` statement inside a `for` loop. In the C language a `break` terminates the execution of the loop that contains it.

parameters:

- `a0`: Address of the first position of a vector `S` containing pointers to strings

return value: None

side effect: The pointers in `S` are now sorted according to the lexicographic order of the strings.



```
LexiSort:
    addi    sp, sp, -32
    sw      ra, 0(sp)
    sw      s0, 4(s0)
    sw      s1, 8(sp)
    sw      s2, 12(sp)
    sw      s3, 16(sp)
    sw      s4, 20(sp)
    sw      s5, 24(sp)
    sw      s6, 28(sp)
    mv      s0, a0          # s0 <- S
    addi    s1, zero, -1    # i <- -1
    addi    s6, zero, -1    # s6 <- -1
Loopi:
    addi    s1, s1, 1       # i <- i+1
    sll     t1, s1, 2       # t1 <- 4*i
    add     t2, s0, t1      # t2 <- Address(S[i])
    lw      t3, 0(t2)       # t3 <- S[i]
    beq     t3, s6, Sorted  # if S[i] == -1 goto Sorted
    addi    s2, s1, -1      # j <- i-1
    blt     s2, zero, Loopi # if j<0 goto Loopi
Loopj:
    sll     t4, s2, 2       # t4 <- j*4
    add     s3, s0, t4      # s3 <- Address(S[j])
    lw      s4, 0(s3)       # Stempj <- S[j]
    lw      s5, 4(s3)       # Stempj1 <- S[j+1]
    mv      a0, s4          # a0 <- Stempj
    mv      a1, s5          # s5 <- Stempj1
    jal     LexiCmp         # Ctemp <- LexiCmp(Stempj,Stempj1)
    ble     a0, zero, Loopi # if Ctemp <= 0 goto Loopi
    sw      s5, 0(s3)       # S[j] <- Stempj1
    sw      s4, 4(s3)       # S[j+1] <- Stempj
    addi    s2, s2, -1      # j <- j-1
    bge     s2, zero, Loopj # if j>=0 goto Loopj
    j       Loopi           # goto Loopi
Sorted:
    lw      ra, 0(sp)
    lw      s0, 4(s0)
    lw      s1, 8(sp)
    lw      s2, 12(sp)
    lw      s3, 16(sp)
    lw      s4, 20(sp)
    lw      s5, 24(sp)
    lw      s6, 28(sp)
    addi    sp, sp, 32
    jalr zero, ra, 0
```

Figure 3: An index-based solution to the `LexiSort` function.

```
LexiSort:
    addi    sp, sp, -28
    sw      ra, 0(sp)
    sw      s0, 4(s0)
    sw      s1, 8(sp)
    sw      s2, 12(sp)
    sw      s3, 16(sp)
    sw      s4, 20(sp)
    sw      s5, 24(sp)
    mv      s0, a0           # s0 <- S
    addi    s1, a0, -4       # T <- S-1
    addi    s2, zero, -1     # s2 <- -1
Loopi:
    addi    s1, s1, 4        # T++
    lw      t1, 0(s1)        # t1 <- *T
    beq     t1, s2, Sorted   # if *T == -1 goto sorted
    addi    s3, s1, -4       # R <- T-1
    blt     s3, s0, Loopi    # if R<S goto Loopi
    lw      s4, 0(s3)        # Rtemp <- *R
    lw      s5, 4(s3)        # Rtemp1 <- *(R+1)
    mv      a0, s4           # a0 <- Rtemp
    mv      a1, s5           # a1 <- Rtemp1
    jal     Lexicmp          # Ctemp <- LexiCmp(Rtemp,Rtemp1)
    ble     a0, zero, Loopi # if Ctemp <= 0 goto Loopi
    sw      s5, 0(s3)        # *R <- Rtemp1
    sw      s4, 4(s3)        # *(R-1) <- Rtemp
    addi    s3, s3, -4       # R--
    bge     s2, s0, Loopj    # if R >= S goto Loopj
    j       Loopi            # goto Loopi
Sorted:
    lw      ra, 0(sp)
    lw      s0, 4(s0)
    lw      s1, 8(sp)
    lw      s2, 12(sp)
    lw      s3, 16(sp)
    lw      s4, 20(sp)
    lw      s5, 24(sp)
    addi    sp, sp, 28
    jalr    zero, ra, 0
```

Figure 4: A pointer-based solution to the `LexiSort` function.