# Assignments week 2

Ethan Bastian, Abdullah Turk

2025-09-05

## Assignment 01 - list reverse

My code:

```cpp
#include <algorithm>
#include <iostream>
#include "linked_list.h"

sax::linked_list_node<std::string> * reverse(sax::linked_list_node<std::string> * unreversed
    if (unreversed == nullptr || unreversed->next ==nullptr) {
        return  unreversed;
    }

    sax::linked_list_node<std::string> * reversed = reverse(unreversed->next);

    unreversed->next->next = unreversed;
    unreversed->next = nullptr;

    return reversed;

}

int main() {
    /* TODO:
        Write a program that reads a singly linked list of strings from standard input,
        reverses the list using recursion, and prints the reversed list to standard output.

        Example input: [apple -> banana -> cherry -> blueberry]
        Example output: [blueberry -> cherry -> banana -> apple]
    */
    sax::linked_list_node<std::string> *head = nullptr;

    std::cin >> head;
```

```
    if (!std::cin) {
        std::cerr << "i failed because of you" << std::endl;
        sax::linked_list_node<std::string>::cleanup(head);
    }

    if (head == nullptr) {
        std::cout << "[]" << std::endl;
        return 0;
    }

    sax::linked_list_node<std::string> * reversed = reverse(head);

    std::cout << reversed << std::endl;
    sax::linked_list_node<std::string>::cleanup(reversed);
    // sax::linked_list_node<std::string>::cleanup(head);




    return 0;
}
```

Time complexity: this algorithm has a time complexity of ....., because ......

## Assignment 03 - quick sort

My code:

```
// fancy algorithm here

#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <fstream>  // for std::ifstream, std::ofstream
#include "counter.h" // for sax::counter

std::vector<std::string>::iterator partition(std::vector<std::string>::iterator begin, std::
    if (begin == end) {
        return begin;
    }

    std::string pivot = *(end - 1);

    auto i = begin;
```

```
    for (auto j = begin; j != end - 1; j++) {
        sax::counter::instance().inc_comparisons();

        if (*j <= pivot) {
            std::swap(*i, *j);

            sax::counter::instance().inc_swaps();

            ++i;
        }
    }

    std::swap(*i, *(end - 1));
    sax::counter::instance().inc_swaps();

    return i;

}

void quick_sort(std::vector<std::string>::iterator begin, std::vector<std::string>::iterator
    if (begin == end || begin +1 == end) {
        return;
    }

    auto pivot_it = partition(begin,end);

    quick_sort(begin, pivot_it);
    quick_sort(pivot_it + 1, end);
}

int main() {
    /* TODO:
        Write a program that reads the names of two files from the standard input,
        reads the strings from the first file into a vector, sorts the vector using quick so
        and then writes the sorted strings to the second file.

        When sorting the strings, the program should count the number of comparisons and swa
        and print these counts to standard output.

        Use the sax::counter class for this (see counter.h and counter.cpp).
    */

    std::string input_filename;
    std::string output_filename;
```

```
    std::cin >> input_filename >> output_filename;

    std::ifstream input(input_filename);
    if (!input) {
        std::cerr << "error reading input_filename" << std::endl;
        return 1;
    }

    std::vector<std::string> words;
    std::string word;
    while (input >> word) {
        words.push_back(word);
    }

    input.close();

    quick_sort(words.begin(), words.end());



    std::ofstream output(output_filename);
    if(!output) {
        std::cerr << "error reading output_filename" << std::endl;
        return 1;
    }

    for(const auto &s : words) {
        output << s << '\n';
    }
    return 0;
}
```

Time complexity: this algorithm has a time complexity of ....., because ......

## Assignment 04 - merge sort

My code:

```
// fancy algorithm here

#include <iostream>
#include <string>
#include <fstream>  // for std::ifstream, std::ofstream
#include "counter.h" // for sax::counter
#include "linked_list.h"
sax::linked_list_node<std::string>* split(sax::linked_list_node<std::string> *head);
```

```cpp
sax::linked_list_node<std::string>* merge_sort(sax::linked_list_node<std::string> *head);
sax::linked_list_node<std::string>* merge(sax::linked_list_node<std::string> *left, sax::lin
int main() {
    std::string input_file, output_file;

    std::cout << "Enter input file name: ";
    std::cin >> input_file;
    std::cout << "Enter output file name: ";
    std::cin >> output_file;

    std::ifstream infile(input_file);
    if (!infile.is_open()) {
        std::cerr << "Error: Could not open input file " << input_file << std::endl;
        return 1;
    }

    sax::linked_list_node<std::string>* head = nullptr;
    sax::linked_list_node<std::string>* tail = nullptr;
    std::string line;

    while (std::getline(infile, line)) {
        auto new_node = new sax::linked_list_node<std::string>();
        new_node->data = line;
        new_node->next = nullptr;

        if (head == nullptr) {
            head = tail = new_node;
        } else {
            tail->next = new_node;
            tail = new_node;
        }
    }
    infile.close();

    sax::counter& counter_instance = sax::counter::instance();
    size_t initial_comparisons = counter_instance.comparisons();

    head = merge_sort(head);

    size_t comparisons_performed = counter_instance.comparisons() - initial_comparisons;

    std::ofstream outfile(output_file);
    if (!outfile.is_open()) {
        std::cerr << "Error: Could not open output file " << output_file << std::endl;
        sax::linked_list_node<std::string>::cleanup(head);
        return 1;
```

```
    }

    auto current = head;
    while (current != nullptr) {
        outfile << current->data << std::endl;
        current = current->next;
    }
    outfile.close();

    std::cout << "Number of comparisons: " << comparisons_performed << std::endl;

    sax::linked_list_node<std::string>::cleanup(head);

    return 0;
}

sax::linked_list_node<std::string>* merge_sort(sax::linked_list_node<std::string> *head) {
    if (head == nullptr || head->next == nullptr) {
        // Base case: list is empty or has one element
        return head;
    }
    // Step 1: Split the list into two halves
    sax::linked_list_node<std::string> *mid = split(head);
    // Step 2: Recursively sort both halves
    head = merge_sort(head);
    mid = merge_sort(mid);
    // Step 3: Merge the sorted halves
    return merge(head, mid);
}
sax::linked_list_node<std::string>* split(sax::linked_list_node<std::string>* head) {
    if (head == nullptr || head->next == nullptr) {
        return nullptr;
    }

    auto slow = head;
    auto fast = head->next;

    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
    }

    auto middle = slow->next;
    slow->next = nullptr;  // Cut the connection

    return middle;
```

```
}

sax::linked_list_node<std::string>* merge(sax::linked_list_node<std::string> *left, sax::lin
    // Create a dummy node to simplify the merge logic
    sax::linked_list_node<std::string> dummy;
    auto current = &dummy;

    while (left != nullptr && right != nullptr) {
        // Count the comparison
        sax::counter::instance().inc_comparisons();

        if (left->data <= right->data) {
            current->next = left;
            left = left->next;
        } else {
            current->next = right;
            right = right->next;
        }
        current = current->next;
    }

    // Attach remaining nodes
    if (left != nullptr) {
        current->next = left;
    } else {
        current->next = right;
    }

    return dummy.next;
}
```

Time complexity: this algorithm has a time complexity of ....., because ......