

CMPT125, Fall 2025

Homework Assignment 3

Due date: Friday, October 31, 2025, 23:59

You need to implement the functions in ***assignment3.c***.
Submit only the ***assignment3.c*** file to CourSys.

Solve all problems in this assignment, one function for each problem.

Grading: The assignment will be graded automatically.

Make sure that your code compiles without warnings/errors, and returns the required output.

Compilation: Your code MUST compile in CSIL with the Makefile provided.

If the code does not compile in CSIL, the grade on the assignment is 0 (zero).

Even if you can't solve a problem, make sure the file compiles properly.

Warnings: Warnings during compilation will reduce points.

More importantly, they indicate that something is probably wrong with the code.

Dynamically allocated arrays: Do not use variable length arrays! Never!

If you need an array of unknown length, you need to use malloc.

Memory leaks: Memory leaks during execution of your code will reduce points.

Make sure all memory used for intermediate calculations are freed properly.

Readability: Your code must be readable, and have reasonable documentation, but not too much. No need to explain `i+=2` with `// increase i by 2`.

Write helper functions if that makes the code more readable.

Testing: An example of a test file is included.

Your code will be tested using the provided tests as well as additional tests.

Do not hard-code any results produced by the functions as we will have additional tests.

You are strongly encouraged to write more tests to check your solution is correct, but you don't have to submit them.

1. You need to implement all the functions in ***assignment3.c***.
2. You should not add `main()` to `assignment3.c`, because it will interfere with `main()` in the test file.
3. Submit only the ***assignment3.c*** file to CourSys.

Problem 1 [40 points, 10 points each]

Write the following four functions:

a) The function gets an array A of length n of ints, and a boolean predicate pred. It returns the first (smallest) index i such that $\text{pred}(A[i]) == \text{true}$. If no such element is not found, the function returns -1.

```
int find(int* A, int n, bool (*pred)(int));
```

b) The function gets an array A of length n of ints, and a boolean predicate pred. It returns the number of indices i such that $\text{pred}(A[i]) == \text{true}$.

```
int count(int* A, int n, bool (*pred)(int));
```

c) The function gets an array A of length n of ints, and a function f. It applies f to each element of A.

```
void map(int* A, int n, int (*f)(int));
```

d) The function gets an array A of length n of ints, and a function f. The function f gets 2 ints and works as follows:

1. Start with accumulator = A[0]
2. For $i=1 \dots \text{length}-1$ compute $\text{accumulator} = f(\text{accumulator}, A[i])$
3. Return accumulator

For example, if f computes the sum of the two inputs, then reduce() will compute the sum of the entire array.

```
int reduce(int* A, int n, int (*f)(int,int));
```

Problem 2 [20 points]

Write a generic implementation of insertion sort.

```
// the function gets an array of length n of objects of given size
// and a compare function
// The function applies Insertion Sort on the array
// using the compare function
// The function returns the number of swaps made by the algorithm
// if compare (a,b)<0, then a must come before b in the sorted array
// if compare (a,b)>0, then a must come after b in the sorted array
int gen_insertion_sort(void* array, int n, size_t size,
                      int (*compare)(const void*, const void*));
```

****Note that the order of the elements in the sorted array depends on the compare function**

Problem 3 [25 points]

Recall the MergeSort algorithm and its merging function.

[15 points] Implement the merging function.

```
// used for MergeSort:
// the function gets an array of length n
// and the index of the midpoint.
// the assumption is that ar[0...mid-1] is sorted
// and ar[mid...n-1] is sorted
// the function merges the two halves into a sorted array
void merge(int* array, int n, int mid);
```

[10 points] Implement the MergeSort algorithm. Use your merge() function as a subroutine.

```
// MergeSort algorithm
void merge_sort(int* array, int n);
```

Problem 4 [15 points]

Write a function that gets an array of strings of length n, and sorts the strings in the array. In the sorted array the strings are arranged as follows:

- Shorter strings need to come before the longer strings
- For two strings of equal length, use the function strcmp() to decide which one needs to come before the other. Specifically, if strcmp(str1, str2)<0, then str1 should be before str2 in the sorted array. if strcmp(str1, str2)>0, then str1 should be after str2 in the sorted array.

* strcmp(str1, str2) compares two strings according to the first non-matching character.

- ❖ **strcmp(str1, str2)>0** if the first non-matching character in str1 is **greater** (in ASCII) than that of str2.
- ❖ **strcmp(str1, str2)<0** if the first non-matching character in str1 is **smaller** (in ASCII) than that of str2.
- ❖ **strcmp(str1, str2)==0** if the two strings are equal.

One way to solve it is to apply qsort() on the array with the corresponding comparison function.

```
void sort_strings(const char* A[], int n)
```

For example:

- Suppose arr = ["Wx", "ab", "Zde", "6_@7h", "7", "hij", "hhh", "b"]. After running sort_strings() arr should become ["7", "b", "ab", "Wx", "Zde", "hhh", "hij", "6787h"]. This is because the ascii value of '7' is smaller than the ascii value of 'b', and the ascii value of 'Z' is smaller than the ascii value of 'h'.