MTH 4300: Algorithms, Computers and Programming II

Spring 2026

Section: SMWA

Problem Set 3

I know that it's very easy to just use AI to solve these problems, but please resist! I expect you all to give these problems an honest attempt. Spend about 30 minutes trying each problem yourself first. If you're still not getting to a good place with your solution, then prompt AI to guide you to a solution rather than expect it to solve the entire problem for you.

Problem 1: In-Place Array Transformation via Pointers

Write a program that reads n integers from the user into a dynamically allocated array, then transforms the array **in place** using pointer-based functions. You must not use array indexing (`arr[i]`) anywhere in your code – use pointer arithmetic and dereferencing instead.

Implement the following functions:

- `void read_array(int* arr, int n)` – reads n integers from the user into the array using pointer arithmetic.
- `void prefix_sum(int* arr, int n)` – transforms the array so that each element becomes the sum of all elements up to and including itself. For example, `{3, 1, 4, 1, 5}` becomes `{3, 4, 8, 9, 14}`. Use a pointer that walks through the array, accumulating the running total.
- `int compact(int* arr, int n)` – removes consecutive duplicates in place and returns the new logical size. Walk two pointers through the array: a **read** pointer that scans every element, and a **write** pointer that only advances when a new distinct value is found. For example, `{1, 1, 3, 3, 3, 2, 2, 1}` becomes `{1, 3, 2, 1}` with a returned size of `4`.
- `void partition_negatives(int* arr, int n)` – rearranges the array so that all negative values appear before all non-negative values. Use two pointers: one starting at the front and one at the back, walking toward each other and swapping when needed. The relative order within each group does not need to be preserved.
- `void print_array(const int* arr, int n)` – prints the array elements separated by spaces, using pointer arithmetic.

Your `main` function should:

1. Prompt the user for n, then allocate an array of size n with `new`
2. Call `read_array` to fill it
3. Print the original array
4. Call `prefix_sum`, then print the prefix sum array
5. Re-read the array (call `read_array` again with the same data or re-prompt – your choice)
6. Call `compact`, then print the compacted array and its new size
7. Re-read the array again
8. Call `partition_negatives`, then print the partitioned array
9. Deallocate the array with `delete[]`

Sample output:

```
Enter n: 8
Enter 8 integers: 1 1 3 3 3 2 2 1

Original:        1 1 3 3 3 2 2 1
Prefix sum:      1 2 5 8 11 13 15 16

Enter 8 integers: 1 1 3 3 3 2 2 1
Compacted:       1 3 2 1
New size:        4

Enter 8 integers: 4 -2 7 -5 0 -1 3 -8
Partitioned:     -8 -2 -1 -5 0 7 3 4
```

Problem 2: Sliding Window Maximum

Given a `std::vector<int>` of n integers and a window size k, compute the maximum value in every contiguous subarray of length k.

Implement the following functions:

- `std::vector<int> read_input(int n)` – reads n integers from the user and returns them in a vector.
- `std::vector<int> sliding_max(const std::vector<int>& nums, int k)` – returns a vector containing the maximum of each window of size k. The result has $n - k + 1$ elements. For example, given `{1, 3, -1, -3, 5, 3}` and $k = 3$, the windows and their maxima are:

```
[1  3  -1] -3  5  3   -> max = 3
 1 [3  -1  -3] 5  3   -> max = 3
 1  3 [-1  -3  5] 3   -> max = 5
 1  3  -1 [-3  5  3]  -> max = 5
```

So the result is `{3, 3, 5, 5}`.

- `void print_vector(const std::vector<int>& v)` – prints the elements separated by spaces.

Your `main` function should:

1. Read n and k from the user
2. Validate that $1 \leq k \leq n$; if not, print `"Invalid window size."` and exit
3. Read the n integers
4. Compute and print the sliding window maxima

A straightforward $O(nk)$ solution (scanning each window for its max) is acceptable.

Sample output:

```
Enter n and k: 6 3
Enter 6 integers: 1 3 -1 -3 5 3
Sliding window maxima: 3 3 5 5
```