April 26, 2021 at 14:04

1. Terminal Sorting.

2. Visualizing sorting algorithms in terminal. This program is trying to visualize the process of sorting a shuffled array of numerical values by applying the Bubble Sort algorithm. (*maybe* more to come)

I originally wrote this code when I was a junior is high school, basically, just for fun. It's been some time since I've even seen this piece of code and I thought it would be great to write it up and make it understandable for *maybe* future maintainers.

3. Most `CWEB` programs share a common structure. This program is no exception. Here, then, is an overview of the file `sort.c` that is defined by this `CWEB` program `sort.w`

⟨ Header files to include 4 ⟩
⟨ Global variables 6 ⟩
⟨ Utility functions 13 ⟩
⟨ Sorting algorithms 22 ⟩
⟨ The main program 7 ⟩

4. We must include libraries to interact with the terminal, allocate and free heap memory, add delay in between "animation frames", and record the program's performance.

⟨ Header files to include 4 ⟩ ≡

```
#include <ncurses.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
```

This code is used in section 3.

5. We also should have some quick definitions, such as delay between our frames in milliseconds. We can set the default value to 0, because the terminal refresh rate itself is a visual bottleneck, which we will talk about later and optimize. If the terminal is too slow and animation is going too fast, we can use `DELAY` to make it go slower and look nicer.

```
#define DELAY 0
```

6. The *ch* variable and *space* just define the actual contents of each frame in our sorting program, such that a single value cell is represented by *ch*, which is a single printable character.

⟨ Global variables 6 ⟩ ≡

```
char *ch = "@";    /* Character to show the "full" cell */
char *space = " ";    /* Character for empty space */
int MAX_X;    /* The maximum number of columns in our terminal window */
int MAX_Y;    /* The maximum number of rows in our terminal window */
```

This code is used in section 3.

7. Now let us talk about the *main* layout.

```
⟨ The main program 7 ⟩ ≡
  int main(void)
  {
    ⟨ Initialize the terminal screen 8 ⟩;
    ⟨ Prepare the array for sorting 9 ⟩;
    ⟨ Run the animation process 10 ⟩;
    ⟨ Show the sorting results 11 ⟩;
    ⟨ Deallocate terminal and array 12 ⟩;
    return OK;
  }
```

This code is used in section 3.

8. For the purposes of manipulating the bits and pieces of the terminal, we use the *ncurses* library. It needs some initial setup with a couple of functions and it can also return us the current dimensions of the caller's terminal!

```
⟨ Initialize the terminal screen 8 ⟩ ≡
  initscr();   /* Initialize the current screen */
  curs_set(0);   /* Set the cursor to zero position */
  keypad(stdscr, TRUE);   /* Allow use of the keyboard */
  noecho();   /* Disable the blinking cursor */
  cbreak();   /* Switch off input buffering */
  getmaxyx(stdscr, MAX_Y, MAX_X);
    /* Set the maximum number of rows and columns into our globals */
```

This code is used in section 7.

9. Sorting is usually applied to some fixed size arrays, where the resulting array's elements will be in descending or ascending order by value. This is no different. For the final picture to look nice, the array will be initialized such that the rate of change of values will be constant. This is handled by convenient *init_arr*. Right after, we will randomly shuffle the array with *shuffle*.

```
⟨ Prepare the array for sorting 9 ⟩ ≡
  int arr_size = MAX_X;
  int *arr = (int *) malloc(sizeof(int) * arr_size);
  init_arr(arr, arr_size, MAX_Y);
  shuffle(arr, arr_size);
```

This code is used in section 7.

10. The animation process would include actually invoking the bubble sort routine and doing some fancy frame updates. Because we also would want our final sorted array to blink on the terminal (indicating that it has been sorted), we will print the array values with a special ANSI escape sequence character.

We would also need to record the performance by simply saving the clock tick value before and after sorting. Their absolute difference divided by the number of clocks per second would give us the total execution time in seconds.

bubble_sort is called only once to sort the passed array with Bubble Sort and its itself calling another function, *update_screen* to efficiently refresh the screen between each swap in the array. More on the reasoning and efficiency later.

print_array prints the passed array with some maximum number of rows and with the last parameter, we can tell *print_array* whether to make the print the array with a blinking effect.

⟨ Run the animation process 10 ⟩ ≡

```
print_array(arr, arr_size, MAX_Y, 0);
clock_t start = clock();
bubble_sort(arr, arr_size);
clock_t end = clock();
print_array(arr, arr_size, MAX_Y, 1);
double exec_time = ((double)(end - start)/CLOCKS_PER_SEC);
```

This code is used in section 7.

11. After running the animation, it's good to put a couple of more lines on the screen, such as showing the sorting algorithm, execution time, etc.

Sequence of *mvprintw* calls simply shows some results, such as number of elements in the array, algorithm (wink, wink, you can modify it to support other algorithms), and the time taken to sort and animate.

The actual placements of those lines is hardcoded for them to be on the top left corner of the screen, it's convenient and doesn't overlap with the blinking array columns.

⟨ Show the sorting results 11 ⟩ ≡

```
mvprintw(4, 1, "Sorting algorithm:            Bubble Sort");
mvprintw(5, 1, "Number of array elements:      %d", arr_size);
mvprintw(6, 1, "Time taken to sort the array:  %f", exec_time);
mvprintw(8, 1, "Written by Sandy Urazayev");
```

This code is used in section 7.

12. After completing the whole sorting process and when animation is done, all we see on our terminal screen is flashing characters, we deallocate the array as we no longer need it. Wait for any of the user's input, block until it happens. Proceed with ending the window session.

⟨ Deallocate terminal and array 12 ⟩ ≡

```
free(arr);
getch();
endwin();
```

This code is used in section 7.

13. Initializing an array with constant rate of change is pretty straightforward. The only quirk is that the absolute lowest value is set to 0 by default.

⟨ Utility functions 13 ⟩ ≡
```
void init_arr(arr, size, max_value)
    int *arr;    /* The actual array to initialize */
    int size;    /* The size of the passed array */
    int max_value;    /* The maximum value in the array */
{
  for (size_t i = 0; i < size; i++) arr[i] = max_value * i/size;
}
```
See also sections 14, 16, 19, and 20.

This code is used in section 3.

14. Shuffling process is no more convoluted, we would need to just generate a random seed so that we can pick random indices in the array, which will get swapped

⟨ Utility functions 13 ⟩ +≡
```
void shuffle(arr, size)
    int *arr;    /* The array that needs to be shuffled */
    int size;    /* The size of the passed array */
{
  srand(time(Λ));    /* Initialize a random seed */
  for (size_t i = 0; i < size; i++) {
    int j = rand() % size;
    ⟨ Swap elements i and j 15 ⟩;
  }
}
```

15. Swapping in C++ can be done with *std*::*swap* from *algorithm*. In C, you can do some XOR magic, define a macro, etc. I'll use a simple temporary variable (yes, it gets reallocated on each iteration, sometimes we write code like this too).

⟨ Swap elements i and j 15 ⟩ ≡
```
int temp = arr[i];
arr[i] = arr[j];
arr[j] = temp;
```
This code is used in section 14.

16. Printing an array is a quick task, for some performance issues and terminal padding I've encountered, notice that elements in the vertical direction, so the number of the printed row goes from top to bottom.

```
⟨ Utility functions 13 ⟩ +≡
  void print_array(arr, size, y, wacky)
      int *arr;    /* The array to print */
      int size;    /* The size of the passed array */
      int y;    /* The maximum value of the array */
      char wacky;    /* Print each cell with a blinking effect */
  {
    if (wacky) {
      ⟨ Turn on bold blinking characters 17 ⟩;
    }
    for (size_t i = 0; i < size; i++)
      for (int j = 0; j < arr[i]; j++) mvprintw(y − j, i, ch);
    ⟨ Turn off bold blinking characters 18 ⟩;
    refresh();
  }
```

17. *ncurses* gives us direct functions to turn on or turn off some ANSI effects of characters that are going to be printed next.

```
⟨ Turn on bold blinking characters 17 ⟩ ≡
  attron(A_BLINK);
  attron(A_BOLD);
```

This code is used in section 16.

18. Similarly for turning off some specific attributes.

```
⟨ Turn off bold blinking characters 18 ⟩ ≡
  attroff(A_BLINK);
  attroff(A_BOLD);
```

This code is used in section 16.

19. Before we overwrite the new swapped value, we first need to clear the columns up. *clear_x* just writes the defined empty character in the range of rows on the specified column.

```
⟨ Utility functions 13 ⟩ +≡
  void clear_x(y, x)
      int y;    /* Number of rows to clear */
      int x;    /* The column number of clear */
  {
    for (int i = 0; i < y; i++) mvprintw(i, x, space);
  }
```

20. Of course, we could print the whole array on every swap and array update, however, this would be incredibly inefficient, as the terminal would need to refresh `MAX_Y` times `MAX_X` characters many many times a second that is not incredibly efficient. Some terminal tearing would occur.

To battle this, we have *update_screen* that only updates a select portion of the screen. Like a buffer update with a diff. It also does in-real-time update of the current number of swaps and comparisons.

A helper function *clear_x* will clear the passed column. Obviously, there are better and more efficient ways to do this, however, I am trying to stay faithful to the code as I wrote it so many years ago. Pretty OK for a guy who just started learning C.

⟨ Utility functions 13 ⟩ +≡

```
void update_screen(y, ind1, ind2, val1, val2, swaps, comps)
    int y;      /* The top value of rows to clear */
    int ind1;      /* The column number of the first swapped element */
    int ind2;      /* The column number of the second swapped element */
    int val1;      /* The value of the first swapped element */
    int val2;      /* The value of the second swapped element */
    int swaps;      /* Number of swaps to show to the user during runtime */
    int comps;      /* Number of comparision to show to the user during runtime */
{
  usleep(100 * DELAY);
  clear_x(y, ind1);
  clear_x(y, ind2);
  mvprintw(1, 1, "Swaps: %d", swaps);
  mvprintw(2, 1, "Comparisons: %d", comps);
  for (int i = 0; i < val1; i++) mvprintw(y - i, ind1, ch);
  attron(A_BOLD);
  for (int i = 0; i < val2; i++) mvprintw(y - i, ind2, ch);
  attroff(A_BOLD);
  refresh();
}
```

21. Bubble Sort.

22. Bubble Sort is probably one of the most foundational sorting algorithms. If you don't know how it works, feel free to search for it and read up on it. The basic principle is that each element in the array is incrementally compared to all succeeding elements and swapped according to the desired order (ascending/descending). Performance is not great, $O(n^2)$, but it's sweet and simple, like first love should be.

update_screen is the *magically-super-fast* function that does the frame update.

⟨ Sorting algorithms 22 ⟩ ≡

```
void bubble_sort(arr, size)
     int *arr;      /* The array to sort */
     size_t size;      /* The size of the passed array */
{
   int swaps = 0, comps = 0;
   for (size_t i = 0; i < size - 1; i++)
     for (size_t j = 0; j < size - i - 1; j++) {
       if (arr[j] > arr[j + 1]) {
          ⟨ Swap elements j and j + 1 23 ⟩;
          swaps++;
       }
       comps++;
       update_screen(MAX_Y, j, j + 1, arr[j], arr[j + 1], swaps, comps);
     }
}
```

This code is used in section 3.

23. Swapping in the bubble sort will be implemented in the simple temp variable way. If further optimization is needed, some other methods can be used. However, the compiler itself should optimize this **very** obvious pattern of instructions.

⟨ Swap elements j and $j + 1$ 23 ⟩ ≡

```
int temp = arr[j];
arr[j] = arr[j + 1];
arr[j + 1] = temp;
```

This code is used in section 22.

24. Index. Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. Error messages are also shown.

Visualizing sorting algorithms in terminal