The BBB sort

The `sort_on_bit` function

```
void sort_on_bit(int* arr, int const length, int const bit_idx);
```

The core recursive function of the algorithm.

- `arr`: array of integers.
- `length`: length of the array `arr`.
- `bit_idx`: the index of the bit on which to do the comparisons for the split.

It splits the array into a `left` and `right` array, then puts every number which has its bit `bit_idx` set to 0 into the `left` array, and every other number (which would thus all have their bit `bit_idx` set to 1) into the `right` array.

Once done, it concatenates them back into the original array, and then recursively calls `sort_on_bit` on the two different parts of the array. We do this instead of calling it recursively on each `left` and `right` array to save up memory.

```
void sort_on_bit(int* arr, int const length, int const bit_idx) {
  // Don't do anything if nbit < 0
  if (bit_idx < 0) return;

  int* left  = malloc(length * sizeof(int));
  int* right = malloc(length * sizeof(int));
  // Keeping track of the number of elements in each array
  int idxl = 0, idxr = 0;

  // Putting values in left and right arrays
  for (int i = 0; i < length; i++) {
    int v = arr[i]; // The current value
    if ((v >> bit_idx) & 1)
      right[idxr++] = v;
    else
      left[idxl++] = v;
  }

  // Putting back values in the original array
  // Note: `idxl` will indicate where in arr the
  // values from the right array start to appear
  for (int i = 0; i < idxl; i++) arr[i] = left[i];
  for (int i = 0; i < idxr; i++) arr[idxl+i] = right[i];
  free(left); free(right); // free the arrays

  // Recursively sort each part of the array on the previous bit
  // We don't need to sort anything if the length is less than 2
  if (idxl > 1) sort_on_bit(arr, idxl, bit_idx-1);
  if (idxr > 1) sort_on_bit(arr+idxl, idxr, bit_idx-1);
}
```

The wrapper function

The `sort_by_bit` function takes the comparison bit as a parameter. We don't want to do that, we want a function which deduces what is the maximum bit index on which to start sorting. Thus, we have this wrapper function, the actual `bbb_sort`:

```
// Simple (and unsafe) max function
int max(int* arr, int const length) {
  int out = arr[0];
  for (int i = 1; i < length; i++)
    if (arr[i] > out) out = arr[i];
  return out;
}

// Wrapper function, actual bbb_sort
void bbb_sort(int* arr, int const length) {
  int m = max(arr, length); // The maximum number
  int msb = m ? log2(m) : 0; // The Most Significant Bit of m
  sort_on_bit(arr, length, msb);
}
```

Complexity of this algorithm

Its complexity is $O(n)$. Indeed, n comparisons are made for each bit index. So more precisely, if k is the maximum number of the array, then the complexity is $O(n \log k)$.

Threading

At every single step of the recursion, the two recursive calls on `sort_on_bit` can be handled in their own separate thread, as they operate on completely independent parts of the array. In fact, each step can be handled in a maximum of $2^{\log k}$ threads.

$$n + \frac{n}{2^1} + \frac{n}{2^2} + \cdots + \frac{n}{2^{\log k}} = n\left(1 + \left(\frac{1}{2}\right)^1 + \left(\frac{1}{2}\right)^2 + \cdots + \left(\frac{1}{2}\right)^{\log k}\right)$$
$$= n\left(1 + \sum_{i=1}^{\log k}\left(\frac{1}{2}\right)^i\right) = n\left(1 + 1 - \left(\frac{1}{2}\right)^{\log k}\right) = n\left(2 - \left(\frac{1}{2}\right)^{\log k}\right)$$
$$= n\left(2 - k^{\log \frac{1}{2}}\right) = n\left(2 - k^{-\log 2}\right) < 2n$$

Taking that into account, the complexity of the algorithm thus becomes $O\left(n\left(2 - k^{-\log 2}\right)\right) \mid k \in \mathbb{N}^+$. Since $n\left(2 - k^{-\log 2}\right)$ is bounded by $2n$, it finally boils down to simply $O(n)$.

Speykious and VOID