Sorting Algorithms Part 2

Anton Gerdelan <gerdela@scss.tcd.ie>

Programming

- More live coding (maybe less theory time)
- Schedule fixed practice for most important skills
  - 30mins/day competent habit, 1hr+ competitive
  - I used to sit with flatmates semi-watching TV with the laptop
  - Code all the elementary sorts
  - Ask questions / discuss / ask for help / feedback
- Join DUCSS and Netsoc - they have some good talks/ workshops

Previously…

- Charts for visualising sorting happening
- Terms: *file, record, key, stable, indirect sort*
- Selection sort - select smallest item from unsorted part (right) of array into current position
- Insertion sort - insert next item into correct position inside sorted part (left) of array (pickup pile and hand of sorted cards)

Bubble Sort Algorithm

- sorted = false
- while (sorted == false)
  - sorted = true
  - loop over data
    - if (next < current)
      - swap(current, next)
      - sorted = false

Bubble Sort

- Time and space complexity? Worst, average, **best**?
- Advantages:
  - Code is simple
  - Can stop early if numbers already sorted
    - No other sorting algorithm does this
    - Can do one run to check before calling complex sorting algorithm
  - "Stable"
- *Sedgewick has a different implementation of Bubble Sort
- Computer scientists have very negative things to say about Bubble Sort's worst case performance vs Insertion Sort.

Summary - Elementary Sorting Algorithms

- Very simple to implement. Also interchangeable. Some useful properties.
- $O(n^2)$ worst case time
  - May not play well with cache - try them with a timer
- $O(1)$ auxiliary memory (1 variable for swapping)
- Stable
- <u>Code all of these yourself as exercise</u>

merge() 2 sorted lists

- I have 2 sorted files (or arrays) A and B - merge them into a new output array
- Create 3 iterators (counters), one per array
  - `int a_index = 0, b_index = 0, output_index = 0;`
- Compare the value at each index, find the smallest
  - copy value to a new array
  - increment counter of the list you copied from
- Add any left-overs from A and B to output

Working Down the Page

List A	List B	Output List	*could track counters too*

3 4 12 | **1** 10 23

3 4 12 | 1 **10** 23 : 1

3 **4** 12 | 1 **10** 23 : 1, 3

3 4 **12** | 1 **10** 23 : 1, 3, 4

… … : …

: 1, 3, 4, 10, 12, 23

Merging

- Fairly simple
- O(N)
- Requires auxiliary memory - how much?
- Should I code this now? Might take a while - error prone.

Merge Sort Algorithm

1. Cut array of keys in half
2. Sort left half (recursively)
3. Sort right half (recursively)
4. Merge the two sorted lists

Merge Sort

- Merging two sorted lists is O(n)
- Bisecting the sort space is O(log(n))
- So the whole sort is O(n * log(n))
- Faster than our O(n^2) elementary sorting algorithms
- More complex to implement
- Auxiliary memory use? O(…)

coding merge() for merge_sort()

- took me over an hour to code correctly
  - always print output and know what result *should* be
- made lots of mistakes and had to use the debugger
  - mixing up index variables
  - using < instead of <=
  - had to create a temp array inside `merge()` to avoid overwriting original data
- simplified my code after looking at others' code
- replacing recursion with loops would be better still

If we have time..

- Coding merge() and merge_sort() — might take too long for lecture - maybe in tutorial?
- One I prepared earlier follows (and link to GitHub in Discussion Board)
- Next: Quicksort, sorting and coding exercise, 2nd lab for assignment.

```c
// first and last are the range of the output list, inclusive
// first half is left list, second half of this is the right list
void merge( int first, int last, int *array ) {
    // make a temporary working array so we don't overwrite our data
    // as we are reading it
    // alloca is dynamic _stack_ memory - freed at function close
    // you could do this with another sort of array or memory
    int* result = alloca(sizeof(int) * (last - first));

   int mid_index = ( first + last ) / 2;
   int left_index = first, right_index = mid_index + 1, output_index = first;

   // compare the lists until one list runs out of list
   while ( left_index <= mid_index && right_index <= last ) {
       if ( array[left_index] < array[right_index] ) {
           result[output_index++] = array[left_index++];
       } else {
           result[output_index++] = array[right_index++];
       }
   }

   // copy any leftovers from either list into output
   // you can probably simplify these into the other loop
   // if you're smarter than me
   while ( left_index <= mid_index ) {
       result[output_index++] = array[left_index++];
   }
   while ( right_index <= last ) {
       result[output_index++] = array[right_index++];
   }

   // copy into original array
   for (int i = first; i <= last; i++) {
       array[i] = result[i];
   }
}
```

```
// declare here so i can recursively call self
void merge_sort( int first_index, int last_index, int* data );

void merge_sort( int first_index, int last_index, int* data ) {
  // break recursion when counters meet in the middle
  if ( first_index >= last_index ) {
    return;
  }
  int mid_index = ( first_index + last_index ) / 2;

  // NB: replacing recursion with loops is usually more efficient
  merge_sort( first_index, mid_index, data );
  merge_sort( mid_index + 1, last_index, data );
  merge( first_index, last_index, data );
}

int main() {
  // create 2 input lists and space for one output list
  int data[] = { 3, 4, 12, 1, 10, 23 }; // initialiser list for array
giving constant values

  // sort with bisections, recursively, from indices 0 to 5, inclusive
  merge_sort( 0, 5, data );

  for ( int i = 0; i < 6; i++ ) {
    printf( "%i ", data[i] );
  }
  printf( "\n" );

  return 0;
}
```