Sorting in OCaml

Sorting algorithms implemented in the OCaml programming language

Clément Pit--Claudel

March 27, 2010

Abstract

In this paper, we present implementations in the OCaml programming language of many popular sorting algorithms, with complexities ranging from quadratic ($O(n^2)$) to logarithmic ($O(n \lg n)$). We then conduce a few tests, in order to evaluate the relative sorting performance of these algorithms. Finally, through careful timing, we demonstrate that our supposedly unoptimized algorithms achieve an approximate 30% performance increase over the OCaml standard library sorting functions.

Keywords: OCaml, Caml Light, Complexity theory, Sorting algorithms, Bubble sort, Insert sort, Selection sort, Merge sort, Quicksort, Heap sort

Introduction The OCaml programming language, despite providing programmers with lots of useful, high-level constructs which drastically reduce development time, is still much of a work-in-progress. Its performance is overall bad, especially when running on Windows systems, and the measurements we present here are a great illustration of how much remains to be done.
When designing the algorithms used in this article, we preferred focusing on simplicity rather than optimization. They do, however, demonstrate excellent performance.

1 Base functions and definitions

```
let swap a b t =
    let t_b = t.(b) in t.(b) <- t.(a); t.(a) <- t_b;;
```

2 Quadratic sorting algorithms

The following algorithms all have $O(n^2)$ time complexity. Space complexity is in all cases $O(n)$. Recursive algorithms (operating on lists) are denoted by a prefixing "r_".

2.1 Bubble sort

```
let bubble_sort t =
  let length = Array.length t in
    for max_pos = length - 1 downto 0 do
      for pos = 0 to max_pos - 1 do
        if t.(pos) > t.(pos + 1) then swap pos (pos + 1) t
      done
    done
;;
```

```
let rec r_bubble_sort l =
  let rec try_swap = function
    | [] -> (false, [])
    | [a] -> (false, [a])
    | a :: b :: tail ->
        if a > b then (true, b :: (snd (try_swap (a :: tail))))
        else let swapped, newlist = try_swap (b :: tail) in (swapped, a ::
            newlist)
  in let rec sort l =
    let swapped, newlist = try_swap l in
      if swapped then sort newlist else newlist
  in sort l
;;
```

2.2 Insert sort

```
let insert_sort t =
  let length = Array.length t in
    for start = 1 to length - 1 do
      let pos = ref start in
        while !pos > 0 && t.(!pos - 1) > t.(!pos) do
          swap (!pos-1) (!pos) t;
          decr pos
        done
     done
;;
```

```
let r_insert_sort l =
  let rec insert value = function
    | [] -> [value]
    | h :: t ->
        if h > value then value :: h :: t
        else h :: insert value t
  in let rec sort = function
    | [] -> []
    | h :: t -> insert h (sort t)
  in sort l
;;
```

2.3 Selection sort

```
let selection_sort t =
  let length = Array.length t in
    for current = 0 to length - 1 do
      let min_id = ref current in
        for pos = current to length - 1 do
          min_id := if (t.(pos) < t.(!min_id)) then pos else !min_id
        done;
        swap current !min_id t;
    done
;;
```

```
let rec r_selection_sort l =
  let rec get_min = function
    | [] -> failwith "Empty"
    | [m] -> m
    | h :: t -> min h (get_min t)
  in let rec rem_min m = function
    | [] -> failwith "Empty"
    | h :: t -> if h = m then t else h :: rem_min m t
  in let rec sort = function
    | [] -> []
    | liste ->
        let min_val = get_min liste in
          min_val :: (sort (rem_min min_val liste))
  in sort l
;;
```

```
let rec r_selection_sort2 l =
  let rec extract = function
    | [] -> failwith "Empty"
    | [a] -> (a, [])
    | head :: tail ->
        let tail_min, newtail = extract tail in
          if tail_min > head then (head, tail_min :: newtail)
          else (tail_min, head :: newtail)
  in let rec sort = function
    | [] -> []
    | liste -> let min, tail = extract liste in min :: sort tail
  in sort l
;;
```

Note For performance reason, we provide two implementations of the recursive selection sort. The first version, although less subtle, demonstrates better performance.

3 Logarithmic sorting algorithms

The following algorithms all have $O(n\log(n))$ average time complexity, although `quicksort` has a worst-case performance of $O(n^2)$ (the others having a worst case of $O(n\log(n))$. Space complexity varies, although usually being $O(n)$. Note that only `quicksort` is an in-place algorithm.

3.1 Merge sort

```
let merge_sort t =
  let merge t1 length1 t2 length2 =
    let out = Array.make (length1 + length2) 0 in
    let pos1, pos2 = ref 0, ref 0 in
      while !pos1 < length1 || !pos2 < length2 do
        let pos = !pos1 + !pos2 in
          if !pos1 = length1 then
            (out.(pos) <- t2.(!pos2); incr pos2)
          else if !pos2 = length2 then
            (out.(pos) <- t1.(!pos1); incr pos1)
          else
            begin
              if t1.(!pos1) < t2.(!pos2) then
                (out.(pos) <- t1.(!pos1); incr pos1)
              else
                (out.(pos) <- t2.(!pos2); incr pos2)
            end;
      done; out;
  in

  let rec sort start length =
    if length = 0 then [||]
    else if length = 1 then [|t.(start)|]
    else
      begin
        let start1 = start and length1 = length / 2 in
        let start2 = start1 + length1 and length2 = length - length1 in
          merge (sort start1 length1) length1 (sort start2 length2) length2
      end
  in
  let size = Array.length t in Array.blit (sort 0 size) 0 t 0 size
;;
```

For the recursive version, we define a `split` function as follows:

```
let split l =
  let rec _split source left right =
    match source, right with
      | [], _ | [_], _ -> (left, right)
      | _ :: _ :: tail, r :: right_tail -> _split tail (r :: left)
          right_tail
      | _ -> assert false
  in _split l [] l;;
```

Note The split function can also be implemented as follows (a little slower though):

```
let split l =
  let rec _split source left right =
    match source with
      | [] -> (left, right)
      | head :: tail -> _split tail right (head :: left)
  in _split l [] [];;
```

```
let r_merge_sort l =
  let rec merge l1 l2 =
    match l1, l2 with
      | [], l | l, [] -> l
      | h1 :: t1, h2 :: t2 ->
          if h1 < h2 then h1 :: (merge t1 (h2 :: t2))
          else            h2 :: (merge (h1 :: t1) t2)
  in

  let rec sort = function
    | ([] | [_]) as sorted -> sorted
    | list -> let left, right = split list in merge (sort left) (sort right
      )
  in sort l
;;
```

```
let tr_merge_sort l =
  let merge l1 l2 =
    let rec _merge l1 l2 result =
      match l1, l2 with
        | [], [] -> result
        | [], h :: t | h :: t, [] -> _merge [] t (h :: result)
        | h1 :: t1, h2 :: t2 ->
            if h1 < h2 then _merge t1 l2 (h1 :: result)
            else            _merge l1 t2 (h2 :: result)
    in List.rev (_merge l1 l2 [])
  in

  let rec sort l merge_fn =
    match l with
      | [] | [_] -> merge_fn l
      | list -> let left, right = split list in
                  sort left (fun leftR -> sort right (fun rightR ->
                    merge_fn (merge leftR rightR)))
  in sort l (fun x -> x)
;;
```

3.2 Quick sort

```
let quicksort t =
  let split start length pivot_pos =
    let pivot = t.(pivot_pos) in
      swap start pivot_pos t;

      let low, high = ref (start + 1), ref (start + length - 1) in
      while !low < !high do
        while !low < !high && t.(!low) <= pivot do
          incr low
        done;

        while !low < !high && t.(!high) >= pivot do
          decr high
        done;

        if !low < !high then swap !low !high t
      done;
      if t.(!low) > pivot then decr low;
      swap start !low t;
      !low;
  in

  let rec sort start length =
    if length > 1 then
      begin
        let pivot_pos = start + Random.int length in
          let new_pos = split start length pivot_pos in
            sort start (new_pos - start);
            sort (new_pos + 1) (start + length - new_pos - 1);
      end
  in sort 0 (Array.length t)
;;
```

```
let r_quicksort l =
  let split list pivot =
    let rec _split inf sup = function
      | [] -> (inf, sup)
      | h :: t ->
          if h < pivot then _split (h :: inf) sup t
          else              _split inf (h :: sup) t
    in _split [] [] list
  in

  let rec sort result = function
    | [] -> result
    | [a] -> a :: result
    | pivot :: t -> let (inf, sup) = split t pivot in sort (pivot :: (sort
       result inf)) sup
  in List.rev (sort [] l)
;;
```

3.3 Heap sort

```
type heap_struct = {data: int array; mutable size: int};;

let heap_sort t =
  let max_size = Array.length t in
  let heap = {data = Array.make (max_size + 1) 0; size = 0} in

  let push x =
    if heap.size = max_size then failwith "Heap overflow";

    heap.data.(heap.size + 1) <- x;

    let pos = ref (heap.size + 1) in
      while !pos <> 1 do
        let newpos = !pos / 2 in
          if heap.data.(newpos) > heap.data.(!pos) then
            swap !pos newpos heap.data;
          pos := newpos;
      done;

      heap.size <- heap.size + 1
  in

  let pop () =
    if heap.size = 0 then failwith "Empty heap";

    let top = heap.data.(1) in
      heap.data.(1) <- heap.data.(heap.size);

      let changed = ref true in
      let pos = ref 1 in
        while !changed && 2 * !pos <= heap.size do
          let minpos = ref !pos in
          let left = 2 * !pos and right = 2 * !pos + 1 in
            if heap.data.(!minpos) > heap.data.(left) then
              minpos := left;
            if right <= heap.size then
              if heap.data.(!minpos) > heap.data.(right) then
                minpos := right;

            changed := not (!pos = !minpos);
            swap !pos !minpos heap.data;

            pos := !minpos;
        done;

        heap.size <- heap.size - 1;
        top;
  in

    for pos = 0 to max_size - 1 do
      push t.(pos)
    done;

    for pos = 0 to max_size - 1 do
      t.(pos) <- pop ()
    done;
;;
```

4 Timing

We have run full measurements of these algorithms performance, which of course confirm the overall better performance of quicksort over other algorithms. We also timed the OCaml library Array.sort and List.sort functions; to our greatest surprise, it was largely outperformed by our simple, unoptimized functions, both when running top-level scripts and compiled code.

4.1 Methodology

We first define functions to generate random input data, for algorithms testing; then, we implement a timing function and, finally, we create a `time_fn` function which operates on a list of input sizes and generates tests for each input size.

```
let random_vect length =
  let t = Array.make length 0 in
    for pos = 0 to length - 1 do
      t.(pos) <- Random.int max_int
    done;
    t;
;;

let rec random_list = function
  | 0 -> []
  | n -> (Random.int max_int) :: (random_list (n-1))
;;

let time randomize sort length =
  let data = randomize length in
  let start = Sys.time () in
    sort data;
    Sys.time () -. start;
;;

let time_fn name random fn counts =
  let rec _time_fn = function
    | [] -> ()
    | count :: tail ->
        Printf.printf "%s : %d elements -> %f\n" name count (time random fn
             count);
        _time_fn tail
  in _time_fn counts; print_newline ()
;;
```

4.2 Testing our sorting algorithms

To test our algorithms, we ran the following commands:

```
time_fn "bubble_sort" random_vect bubble_sort [10; 100; 1000; 10000];
time_fn "insert_sort" random_vect insert_sort [10; 100; 1000; 10000];
time_fn "selection_sort" random_vect selection_sort [10; 100; 1000; 10000];
time_fn "quicksort" random_vect quicksort [10; 100; 1000; 10000; 100000; 1000000];
time_fn "merge_sort" random_vect merge_sort [10; 100; 1000; 10000; 100000; 1000000];
time_fn "heap_sort" random_vect heap_sort [10; 100; 1000; 10000; 100000; 1000000];
time_fn "Array.sort" random_vect (Array.sort compare) [10; 100; 1000; 10000; 100000; 1000000];;

time_fn "r_bubble_sort" random_list r_bubble_sort [10; 100; 1000; 10000];
time_fn "r_insert_sort" random_list r_insert_sort [10; 100; 1000; 10000];
time_fn "r_selection_sort" random_list r_selection_sort [10; 100; 1000; 10000];
time_fn "r_selection_sort2" random_list r_selection_sort2 [10; 100; 1000; 10000];
time_fn "r_quicksort" random_list r_quicksort [10; 100; 1000; 10000; 100000; 1000000];
time_fn "r_merge_sort" random_list r_merge_sort [10; 100; 1000; 10000; 100000; 1000000];
time_fn "tr_merge_sort" random_list tr_merge_sort [10; 100; 1000; 10000; 100000; 1000000];
time_fn "List.sort" random_list (List.sort compare) [10; 100; 1000; 10000; 100000; 1000000];;
```

4.3 Results

On a Intel Pentium 4 CPU, 3.20 GHz machine running Windows 7, we obtained the following results (all times are expressed in seconds):

```
bubble_sort : 10 elements -> 0.000000
bubble_sort : 100 elements -> 0.002000
bubble_sort : 1000 elements -> 0.234000
bubble_sort : 10000 elements -> 21.111000

insert_sort : 10 elements -> 0.000000
insert_sort : 100 elements -> 0.002000
insert_sort : 1000 elements -> 0.189000
insert_sort : 10000 elements -> 13.980000

selection_sort : 10 elements -> 0.000000
selection_sort : 100 elements -> 0.001000
selection_sort : 1000 elements -> 0.130000
selection_sort : 10000 elements -> 10.368000

quicksort : 10 elements -> 0.000000
quicksort : 100 elements -> 0.001000
quicksort : 1000 elements -> 0.006000
quicksort : 10000 elements -> 0.098000
quicksort : 100000 elements -> 0.939000
quicksort : 1000000 elements -> 11.571000

merge_sort : 10 elements -> 0.000000
merge_sort : 100 elements -> 0.000000
merge_sort : 1000 elements -> 0.007000
merge_sort : 10000 elements -> 0.113000
merge_sort : 100000 elements -> 1.104000
merge_sort : 1000000 elements -> 13.133000

heap_sort : 10 elements -> 0.000000
heap_sort : 100 elements -> 0.001000
heap_sort : 1000 elements -> 0.011000
heap_sort : 10000 elements -> 0.210000
heap_sort : 100000 elements -> 2.168000
heap_sort : 1000000 elements -> 28.454000

Array.sort : 10 elements -> 0.000000
Array.sort : 100 elements -> 0.000000
Array.sort : 1000 elements -> 0.008000
Array.sort : 10000 elements -> 0.097000
Array.sort : 100000 elements -> 1.221000
Array.sort : 1000000 elements -> 18.383000
```

```
r_bubble_sort : 10 elements -> 0.000000
r_bubble_sort : 100 elements -> 0.006000
r_bubble_sort : 1000 elements -> 0.431000
r_bubble_sort : 10000 elements -> 77.651000

r_insert_sort : 10 elements -> 0.000000
r_insert_sort : 100 elements -> 0.001000
r_insert_sort : 1000 elements -> 0.105000
r_insert_sort : 10000 elements -> 8.048000

r_selection_sort : 10 elements -> 0.000000
r_selection_sort : 100 elements -> 0.002000
r_selection_sort : 1000 elements -> 0.258000
r_selection_sort : 10000 elements -> 22.524000

r_selection_sort2 : 10 elements -> 0.000000
r_selection_sort2 : 100 elements -> 0.003000
r_selection_sort2 : 1000 elements -> 0.280000
r_selection_sort2 : 10000 elements -> 27.43800

r_quicksort : 10 elements -> 0.000000
r_quicksort : 100 elements -> 0.000000
r_quicksort : 1000 elements -> 0.004000
r_quicksort : 10000 elements -> 0.243000
r_quicksort : 100000 elements -> 1.011000
Stack overflow during evaluation (...).

r_merge_sort : 10 elements -> 0.000000
r_merge_sort : 100 elements -> 0.001000
r_merge_sort : 1000 elements -> 0.006000
r_merge_sort : 10000 elements -> 0.209000
r_merge_sort : 100000 elements -> 1.535000
Stack overflow during evaluation (...).

tr_merge_sort : 10 elements -> 0.000000
tr_merge_sort : 100 elements -> 0.002000
tr_merge_sort : 1000 elements -> 0.010000
tr_merge_sort : 10000 elements -> 0.151000
tr_merge_sort : 100000 elements -> 2.082000
Stack overflow during evaluation (...).

List.sort : 10 elements -> 0.000000
List.sort : 100 elements -> 0.000000
List.sort : 1000 elements -> 0.005000
List.sort : 10000 elements -> 0.145000
List.sort : 100000 elements -> 1.380000
Stack overflow during evaluation (...).
```

5 Conclusion

The library sorting functions implementation appears to be deficient.

Compared to the theoretic $O(n \lg(n))$, the library Array.sort function requires an approximate $36.8 \cdot n \lg(n)$ operations to complete, assuming an approximate $40 \cdot 10^6$ operations per second, while our own functions need $23.2 \cdot n \lg(n)$, $26.3 \cdot n \lg(n)$, and $57 \cdot n \lg(n)$ operations respectively. Heap sort, although being a very easy to implement algorithm, demonstrates bad performance, while both quicksort and merge sort outperform the OCaml library function Array.sort (by an approximate 36% for the former, and 28% for the later).

Similarly, recursive merge sort demonstrates disappointing performance ($37 \cdot n \lg(n)$), whereas recursive quicksort achieves overall good performance ($24.3 \cdot n \lg(n)$), outperforming the library List.sort function ($33.2 \cdot n \lg(n)$) by an approximate 26%.

Identical results were obtained when running compiled versions of these functions. Note that due to stack overflow exceptions – occurring as a result of using non-tail-recursive algorithms –, constants were evaluated based on 10^6 elements sets in the array sorting case, whereas 10^5 elements sets were used in the list sorting case.