3: Sorting: Sample

Sample Solution
23rd Aug 2025
Collaborators: None

1: "Sort an array using Bubble Sort"

BLUEPRINT

Requires:

List l : 'a list

myCompareFunction : 'a -> 'a -> bool (myCompareFunction x y returns true iff x is considered 'greater' than y in the required ordering)

Ensures:

$\forall i \in [0, length(l) - 1], \forall j \in [0, length(l) - 1]$, if $i < j$ then $myCompareFunctionl[i]l[j] = false$

Time Complexity: $O(n^2)$

Space Complexity: $O(n)$ (A new list is created in each pass because of immutability in OCaml)

Input: l = [5; 3; 8; 6; 2]
Output: [2; 3; 5; 6; 8]
Input: l = ["banana"; "apple"; "cherry"]
Output: ["apple"; "banana"; "cherry"]
Input: l = [True; False; True]
Output: [False; True; True]

STEPS

Step 1: We need to define a function that defines ordering for elements of the list i.e. the basis for performing a comparison and deciding if compare A B = true or false.

Step 2: Bubble sorting means we will repeatedly swap adjacent elements if they are in the wrong order. So, we can begin by iterating over the entire array once. In this iteration, if any 'inversions' are observed, swap them.

Step 3: At the end of the first pass, the largest element is at the correct place. So iterating till all indexes from 0 to n-2 is enough. In the i^{th} pass, the i^{th} largest element is in the correct place. So we set our bound for iterating over the list accordingly to n-i .

OCAML CODE ICS Verified

```
(* To sort, we need a comparison function that defines ordering for elements *)
let myCompareFunction (x : int) (y : int) =
    x > y (* This can be changed to the desired comparison logic, change the type signature of the function accordingly *)
;;

(* Swap elements based on a given comparison function *)
let swap (x : 'a) (y : 'a) (compareFunction : 'a -> 'a -> bool) : 'a * 'a =
    match compareFunction x y with
    | true -> (y, x)
    | false -> (x, y)
;;

(* Pass a given list, swapping every element pairwise using the swap function *)
let rec pass (lst : 'a list) (index : int) : 'a list =
```

```
    match index with
    | 0 -> lst
    | _ ->
      (match lst with
      | [] -> []
      | [x] -> [x]
      | x :: y :: xs -> let (a, b) = swap x y myCompareFunction in a :: pass (b :: xs) (index -1))
;;

(* Iterate over the list n-1 times, stopping at n-1, n-2, .... 2 each time *)
let rec bubble_sort_helper (lst : 'a list) (l : int) : 'a list =
    match l with
    | _ when l <= 0 -> lst
    | _ -> bubble_sort_helper (pass lst (l)) (l-1)
;;

let bubble_sort (lst : 'a list) : 'a list =
    let length = List.length lst in
    bubble_sort_helper lst (length - 1)
;;

(* myCompareFunction can also be passed as a parameter to relevant functions as an alternative design choice. Here, for concision of nota
```

PROOF

INDUCTION

Base Case:

For a list of length 0, the list is empty, therefore trivially sorted.

bubble_sort lst = bubble_sort_helper lst -1 0 = lst

The function is correct for a list of length 0.

Inductive Hypothesis:

Assume the function works for lists of length n, denoted by l_n (meaning it sorts the first n elements correctly).

Inductive Step:

For a list of length n+1 l_{n+1},

bubble_sort_helper lst (n) 0 = bubble_sort_helper (pass lst n) (n-1)

In pass lst n, we compare the first two head elements and place them in their correct positions with respect to each other in the line

(let (a, b) = swap x y myCompareFunction in a :: pass (b :: xs)),

then we call pass (b :: xs) (n - 1).

This is done till index = 0 (indicating that we consider elements from 0 to n because index was originally n),

comparing all the elements, ensuring the element that is largest is at the end of the list (because this is the element that 'wins' all comparisons).

Then we return to bubble_sort_helper (pass lst n) (n-1), which is a call for l_n, which we assumed works correctly by the inductive hypothesis.

Therefore, function works correctly for l_{n+1}. The function is correct for all inputs.