Marian Willard and Caroline Danzi
Gregory Gelfond
CSE 464 Algorithms
4 March 2016

Selection Sort

Pseudo code

```
1     public ssort(int[] nums){
2           if(nums.length < 2){
3                 return nums
4           }
5           for i = 0 to nums.length{
6                 min = minimum(nums, i)
7                 swap(nums, i, min)
8           }
9           return nums
10    }
```

1. Function Specifications

`ssort`: If `nums` is any sequence of integers, then `ssort(nums)` will return the sequence `nums` with its elements sorted in ascending order.

`minimum`: If `nums` is any sequence of integers, then `minimum(nums, i)` returns the index of the smallest element in the interval `[i, nums.length)`, where `i` is a legal index into the array `nums`. The smallest element is defined as an integer x such that x is less than all other integers in the sequence from index `i` to the last element of the sequence.

`swap`: If `nums` is any sequence of integers and `i` and `j` are legal indices into `nums`, then `swap(nums, i, j)` puts the element at index `i` in index location `j` and the element that was at index location `j` into the index location `i`.

2. Loop Conditions for the loop at line 5

Termination condition: The loop terminates when i equals the length of the sequence. Since the length of the sequence is always finite, the loop will always terminate.

Loop invariant: the elements in `nums` on the interval `[0, i)` are correctly sorted and are less than or equal to the elements in the legal index range `[i, nums.length)`.

3. Sketch of the Proof of Correctness

Assumptions: The functions `minimum` and `swap` have been shown to function as specified above.

Claim: For any sequence of integers `nums`, `ssort(nums)` will modify `nums` so that the elements are sorted in ascending order.

Proof:

If the sequence has length less than or equal to 1, the sequence is trivially sorted. Otherwise:

1) The loop terminates when i (the current index) is equal to the length of the array. Since the sequence is always finite, the loop will always terminate.
2) Prove the loop invariant: *The elements in nums on the interval [0, i) are correctly sorted in ascending order*.
    a) Prior to entering the loop: i = 0, so the interval [0, i) is empty (has length 0). A sequence of length 0 is trivially sorted, so the invariant is true before entering the loop body.
    b) Assume the invariant is true after every iteration k, where $0 \leq k \leq i$.
    c) Consider iteration k + 1: All elements on the range [0, i-1) are correctly sorted from the previous iteration k. On line 6, we find the index of the minimum element in the index range [i, nums.length). On line 7 we then swap the element at this position with the element at i, thus ensuring that at index i we have the smallest element remaining in the unsorted range. This element will be greater than all preceding elements in the range [0, i) and less than all those following in the range [i+1, nums.length).
3) From (1), the loop terminates when i equals the length of the array. From (2) we see that the elements in the range [0, i) are correctly sorted. Thus, when the loop terminates and i = nums.length, the elements on the interval [0, nums.length) will be correctly sorted. Since the range [0, nums.length) is the entire array, the claim is true.

4. Analysis of Computational Complexity

Selection sort has a worst-case computational complexity of $O(n^2)$, where n is the length of the sequence to sort. In our implementation, we used arrays to represent the sequences. This allowed for constant-time element lookup, as opposed to linear time (which is common for data structures such as linked lists). Thus, the swap function could be performed in constant time.

In the ssort function, there is a loop that iterates over all elements in the sequence, so this action is performed in linear time. Additionally, there is a loop in the function minimum that iterates over all elements in the range [i, nums.length). The loop in minimum is bounded by O(n) as well. So for every iteration of the outer loop, we are performing an operation that takes at worst O(n) time. Thus, the total worst-case performance time for the entire selection sort algorithm is $O(n*n) = O(n^2)$ time (quadratic time).

Merge Sort

Note: Merge sort was invented by John Von Neumann (source: Wolfram MathWorld).

Pseudo Code

```
      define msort(int[] nums) {
            if(nums.length > 1) {
                  int mid = nums.length / 2
                  left = nums[0..mid]
                  right = nums[mid..nums.length]
a                 msort(left)
b                 msort(right)
c                 merge(nums, left, right)
            }
      }

      define merge(int[] nums, int[] left, int[] right) {
            int i = 0
            int j = 0
            int x = 0

1           loop until all the elements in left or all the elements in
            right are fully copied to nums:
                  if left[i] < right[j]:
                        nums[x] = left[i]
                        i++
                  else
                        nums[x] = right[j]
                        j++
                  x++

2           while left still has elements:
                  nums[x] = left[i]
                  i++
                  x++

3           while right still has elements:
                  nums[x] = right[j]
                  j++
                  x++
      }
```

1. Function Specifications

`msort`: For any sequence of integers `nums`, `msort(nums)` will modify `nums` so that its elements are sorted in ascending order.

`merge`: For sequences of integers `nums`, `left`, and `right`, where `left` and `right` are sorted in ascending order, `merge(nums, left, right)` will modify `nums` to contain the elements of both `left` and `right` sorted in ascending order.

2. Loop Conditions

1) *Termination*: The loop terminates when either all the elements from `left` are copied to `nums` or all the elements from `right` are copied to `nums`. This is indicated by `i = left.length` or `j = right.length`. Since either `i` or `j` is increased by one each time through the loop, the loop will always terminate.
   Loop invariant: `nums` contains all the elements of `left` in the index range `[0, i)` and all the elements of `right` in the index range `[0, j)` sorted in ascending order.

2) *Termination*: The loop terminates when the pointer `i` equals `left.length`. Since `i` is increased by one each time through the loop, the loop will always terminate.
   Loop invariant: The elements in `left` from `[0, i)` appear in `nums` in the same order they appear in `left`.

3) *Termination*: The loop terminates when the pointer `j` equals `right.length`. Since `j` is increased by one each time through the loop, the loop will always terminate.
   Loop invariant: The elements in `right` from indices `[0, j)` appear in `nums` in the same order that they appear in `right`.

3. Sketch of the Proof of Correctness

Prove the claim:

Φ `msort`: For any sequence of integers `nums`, `msort(nums)` will modify `nums` so that its elements are sorted in ascending order.

Proof by Induction on the length of the sequence `nums`:

Assumptions: `merge` has been shown to function correctly as specified above.

Base case: If the length of the sequence is less than or equal to 1, the sequence is trivially sorted.

Inductive Hypothesis: For any sequence of integers `nums` of length ≤ N where N is a natural number, `msort(nums)` functions as specified in Φ.

Prove for a sequence of integers of length N + 1: Consider sequence of length N + 1

1) In the line marked a, we recursively sort the left half of the sequence, which has a length that is less than or equal to (N + 1) / 2
2) In the line marked b, we recursively sort the right half of the sequence, which has a length that is less than or equal to (N + 1) / 2
3) Since (N + 1) / 2 is less than N, from our assumption we know that the left and right halves of the sequence will be correctly sorted. Then, on the line marked c we know from the claim for merge that the left and right halves will be correctly merged into a sorted sequence, thus satisfying the claim that msort correctly sorts a sequence of integers.

4. Analysis of Computational Complexity

The computational complexity of merge sort is O(n log n), where n represents the length of the sequence. At every step of the recursion, we decrease the input size by a factor of two, so this part of the algorithm is O(log n). The merge function is linear because we traverse the entirety of the search space - the length of left plus the length of right - once. Since log n merges are executed, and each merge takes O(n) time, so the overall computation complexity is O(n * log n).