MA407: Algorithms and Computation
Exercise 2

October 18, 2024

Jay Razdan

---

1. **Exercise 2.1**
   (a) What does **process(A)** do to a list **A**?
   (b) Show that the inner **for** loop in lines 4–6 satisfies the following loop invariant:

$$A[m] \geq A[x], \quad \text{for all x = i,..., j-1.} \tag{1}$$

   Use this to make a conclusion about m after execution of line 6 in the last iteration of the inner **for** loop for a fixed value of **i**.
   (c) Formulate a loop invariant for the outer **for** loop in lines 2-7 and use your loop invariant together with part (b) to prove that **process(A)** does what you stated in part (a).
   (d) Describe the best- and the worst-case running time of the algorithm for a list **A** of length n in an asymptotic sense or, even better, in big-O notation.

Ans:

(a) **process(A)** is a function that takes as an input, a list A of integers, and subsequently sorts the list in descending order i.e. indexed from greatest to lowest integer.

(b) This loop invariant says that at the start of each iteration of the inner **for** loop, the value indexed at m is the maximum value in the sub-array $A[i, ..., j-1]$. By means of finite induction, we can prove that this loop invariant holds throughout the execution of the inner **for** loop.

1. **Initialization**

Before the inner **for** loop begins, $j = i + 1$ and $m = i$.

Sub-array: $A[i, ..., j-1] = A[i, ..., i] = A[i]$

Since there is only a single element in the sub-array, it follows that $A[m] = A[i]$, thus proving the **loop invariant is true prior to the first iteration of the loop**.

2. **Maintenance**

Inductive hypothesis: Loop invariant holds at the start of some iteration j of the inner **for** loop, i.e. for some $j > i+1$, $A[m] \geq A[x]$, for all x = i,..., j-1.

During the iteration, the algorithm checks if $A[m] < A[j]$ and we increment j by 1.

Sub-array: $A[i, ..., (j+1)-1] = A[i, ..., j]$

If true, it will update the index m to equal the current index of j. So, the value at index m will equal the value at index j, i.e. $A[m] = A[j]$. Therefore, $A[m]$ is the maximum value in the sub-array because not only is it greater than the value at the current index of j, as per the inductive hypothesis it is also greater than the previous maximum of $A[m]$ and subsequently all values in $A[i, ..., j-1]$.

Otherwise, since $A[m] \geq A[j]$, the index m is not updated and remains the index of the maximum value in the sub-array $A[i, ..., j-1]$ (as per the inductive hypothesis), and therefore the current sub-array $A[i, ..., j]$.

In either case, it has been shown that **if the loop invariant is true before an iteration of the loop, it remains true before the next iteration**.

3. **Termination**

The inner **for** loop terminates when $j = len(A)$.

Sub-array: $A[i, ..., j-1] = A[i, ..., len(A)-1]$

As per the maintenance step, since it has been shown that for some $j \geq i+1$, $A[m] \geq A[x]$, for all x = i,..., j-1, when the loops ends, $A[m]$ is the maximum element in the final sub-array $A[i, ..., len(A)-1]$.

Hence, it has been shown that the inner **for** loop of the **process(A)** function satisfies the proposed loop invariant.

(c) **Potential outer loop invariant**: At the start of each iteration i of the outer **for** loop, the sub-array $A[0, .., i-1]$ contains the i largest elements of the array, sorted in descending order. Additionally, the remaining unsorted elements of the array are contained within the sub-array $A[i, ..., len(A)-1]$.

By means of finite induction, we can prove that this loop invariant holds true throughout the execution of the algorithm, in order to prove its correctness.

1. **Initialization**

Before the outer **for** loop begins, $i = 0$.

Sub-array: $A[0, ..., i-1] = A[0, ..., -1] = A[]$
Remaining Sub-array: $A[i, ..., len(A)-1] = A[0, ..., len(A)-1]$

Since the sub-array is empty and sorted by default, and the remaining sub-array contains the array itself which is unsorted, the outer **for** loop satisfies the loop invariant such that **prior to the first iteration, the loop invariant is true**.

2. **Maintenance**

Inductive hypothesis: Loop invariant holds at the start of some iteration $i > 0$.

During the iteration of the outer **for** loop, we increment i by 1 and the algorithms uses the inner **for** loop to find the index m of the maximum element in the remaining sub-array $A[i+1, ..., len(A)-1]$, and then swaps the value indexed at the current value of i with this maximum element.

Sub-array: $A[0, ..., (i+1)-1] = A[0, ..., i]$
Remaining Sub-array: $A[i+1, ..., len(A)-1]$

Since we have proved the correctness of the inner **for** loop, it is known that the index m at the end of the loop will point to the maximum value of the remaining sub-array. As a result, the above sub-array will contain the largest i elements of the array. Additionally, as per the inductive hypothesis, we know that because the loop invariant holds before the start of this iteration, the prior sub-array $A[0, ..., i-1]$ is sorted is descending order, and so before the start of the next iteration, the sub-array $A[0, ..., i]$ will also be sorted in a such a manner.

Then, again by means of the inductive hypothesis, since before the start of this iteration, the prior remaining sub-array contained the remaining unsorted elements of the array, the algorithm during this iteration also satisfies that the current remaining sub-array contains the remaining unsorted elements of array $A[i+1, ..., len(A)-1]$.

Thus, it has been shown that **if the loop invariant is true before an iteration of the outer for loop, it remains true before the next iteration**.

3. **Termination**

The outer **for** for loop terminates when $i = len(A) - 1$.

Sub-array: $A[0, ..., (len(A)-1)-1] = A[0, ..., len(A)-2]$
Remaining Sub-array: $A[len(A)-1, ..., len(A)-1] = A[len(A)-1]$

As per the maintenance step, since it has been shown that for some $i > 0$, the loop invariant holds, when the outer **for** loop ends, it is known that the above sub-array contains all the largest $i = len(A) - 1$ elements of the array

(sorted in descending order), and that the remaining sub-array contains, by default, the remaining last, and smallest element of the array.

Hence, when the outer **for** loop terminates, the entire array $A[0, ..., len(A) - 1]$ is sorted in descending order, thus proving what was stated in the answer to part (a).

(d) Given an input list A of length n, regardless of whether A is already ideally sorted in descending order or contrarily in ascending order, the algorithm will have to iterate through both the inner and outer loops $n - 1$ times. Therefore, the number of permutations the algorithm will have to consider is equal to $(n - 1)^2$. In terms of an asymptotic sense, and with regards to big-O notation, since constant terms diminish, the asymptotic running-time of this algorithm would be $O(n^2)$.

2. **Exercise 2.2**

Describe a $\Theta(n \log n)$-time algorithm that, given a list S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x. Prove the correctness of your algorithm using a loop invariant.

Ans:

We can first sort the list S in ascending order using an algorithm that runs in $\Theta(n \log n)$-time such as the **MergeSort**, and then use pointers on both ends of the sorted list, moving inwards if the sum of the values at the pointers is less than or greater than the required value of x, respectively - otherwise, the sum is equal to the required value of x and the algorithm return True. If by the end of the loop such a sum is not found, then the algorithm will return False.

So, the overall asymptotic Theta time-complexity for this algorithm is
$:= \Theta(n \log n) + \Theta(n) = \Theta(n \log n)$

Here is the pseudo-code for the above outlined algorithm:

```
1:  procedure TwoElementsSumOrNot(S, x)
2:      Call MergeSort(S)
3:      left ← 0
4:      right ← len(S) − 1
5:      while left < right do
6:          current_sum ← S[left] + S[right]
7:          if current_sum = x then
8:              return True
9:          else if current_sum < x then
10:             left ← left + 1
11:         else
12:             right ← right − 1
13:         end if
14:     end while
15:     return False
16: end procedure
```

To prove to the correctness of the algorithm, I will assume the correctness of the **MergeSort** algorithm as it is well-documented and tested, and therefore unnecessary to show, in this case. In other words, to show the correctness of this algorithm, it will be shown that the **while** loop in lines 5-14 satisfies a loop invariant.

Potential while loop invariant: At the start of each iteration of the **while** loop, if there indeed exist two elements of the sub-array $S[left, ..., right]$ that sum to x, then, at the very least, one of them must be $S[left]$ or $S[right]$.

1. **Initialization**

Before the **while** loop begins, $left = 0$ and $right = len(S) - 1$.

Sub-array: $S[left, ..., right] = S[0, ..., len(S) - 1] =:$ Array

The algorithm checks if the elements at the extreme ends of array sum to x, and if this is false, then the pointers are adjusted accordingly - in this case, the **while** loop satisfies the loop invariant such that **prior to the first iteration, the loop invariant is true**.

2. **Maintenance**

Inductive hypothesis: Assume loop invariant holds true before the start of some iteration, for when $left < right$.

The algorithm computes the sum of the values corresponding to where the left and right pointers currently are. The first conditional will return **True** if this sum is indeed equal to the required value of x.

The else-conditional will increment the left pointer by 1 if this sum is less than the required value of x, because the list S is sorted in ascending order. Therefore, as per the inductive hypothesis, a valid pair, if any, must exist in this updated sub-array.

The final else-conditional will increment the right pointer by -1 if this sum is greater than the required value of x, because the list S is sorted in ascending order. Therefore, again, as per the inductive hypothesis, a valid pair, if any, must exist in this updated sub-array.

Thus, it has been shown that **if the loop invariant is true before an iteration of the while loop, it remains true before the next iteration**.

3. **Termination**

The **while** loop terminates when $left >= right$ i.e. before the left and right pointers cross over each other.

Since, at this point, all the possibilities of sums within the sub-array have been used up i.e. no pair summing to the required value of x has been found. As a result, the algorithm will return **False**, correctly identifying that no such pair of integers in S summing to x exist.

Hence, the correctness of the above outlined algorithm has been proven and shown via a loop invariant that is satisfied by the **while** loop.