Exercise 4.3 One-Looped Sort (1 point)

Consider the following pseudocode whose goal is to sort an array A containing n integers.

Algorithm 1 Input: array $A[0 \dots n-1]$

```
1: i ← 0
2: while i < n do
3:     if i = 0 or A[i] ≥ A[i − 1] then
4:         i ← i + 1
5:     else
6:         swap A[i] and A[i − 1]
7:         i ← i − 1
8:     end if
9: end while
```

(a) Show the steps of the algorithm on the input $A = [10, 20, 30, 40, 50, 25]$ until termination. Specifically, give the contents of the array A and the value of i after each iteration of the while loop.

Solution:

Initial array: $A = [10, 20, 30, 40, 50, 25]$.

- $i = 1$, $A = [10, 20, 30, 40, 50, 25]$
- $i = 2$, $A = [10, 20, 30, 40, 50, 25]$
- $i = 3$, $A = [10, 20, 30, 40, 50, 25]$
- $i = 4$, $A = [10, 20, 30, 40, 50, 25]$
- $i = 5$, $A = [10, 20, 30, 40, 50, 25]$
- $i = 4$, $A = [10, 20, 30, 40, 25, 50]$
- $i = 3$, $A = [10, 20, 30, 25, 40, 50]$
- $i = 2$, $A = [10, 20, 25, 30, 40, 50]$
- $i = 3$, $A = [10, 20, 25, 30, 40, 50]$
- $i = 4$, $A = [10, 20, 25, 30, 40, 50]$
- $i = 5$, $A = [10, 20, 25, 30, 40, 50] \rightarrow$ terminates when $i = 6 = n$.

(b) Explain why the algorithm correctly sorts any input array. Formulate a reasonable loop invariant, prove it (e.g., using induction), and conclude using that invariant that the algorithm correctly sorts the array.

Hint: Use the invariant "at the moment when the variable i gets incremented to a new value $i = k$ for the first time, the first k elements of the array are sorted in increasing order."

Solution:

The algorithm sorts an array by letting a pointer run through every one of its elements and comparing it each time with the previous. If the current pointer's element is smaller than the previous, both are swapped and the pointer decreases until the correct place for the element is found.

Invariant: At the moment when the variable i gets incremented to a new value $i = k$ for the first time, the first k elements of the array are sorted in increasing order.

Initialization: The invariant holds for the first iteration. When $i = 1$, the first element is always sorted.

Maintenance: The invariant holds for any arbitrary start of an iteration, because the variable i gets decremented as many times as necessary until every element that needed to be swapped was successfully swapped, before incrementing i.

Termination: The algorithm stops as soon as the variable i gets incremented to be equal to the length of the array. At this point, the invariant guarantees that the array has been correctly sorted.

(c) Give a reasonable running-time upper bound, expressed in O-notation.

Solution:

This algorithm has its *worst-case* when, before every increase of i, it has to perform the most amount of swaps and comparisons. Take for example an array sorted in descending order: before every increment of i, a linear number of swaps and comparisons must be made. This is in the order of $C \cdot i$, where C is a constant. We can ignore this constant factor, meaning the sum of all comparisons and swaps performed in an array of length n would look something like:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \leq O(n^2)$$

Giving us our upper bound.

Exercise 4.4 Searching for the summit (1 point)

Suppose we are given an array $A[1 \dots n]$ with n unique integers that satisfies the following property. There exists an integer $k \in [1, n]$, called the *summit index*, such that $A[1 \dots k]$ is a strictly increasing array and $A[k \dots n]$ is a strictly decreasing array. We say an array is *valid* if it satisfies the above properties.

(a) Provide an algorithm that finds this k with worst-case running time $O(\log n)$. Give the pseudocode and an argument why its worst-case running time is $O(\log n)$.

Note: Be careful about edge cases! It could happen that $k = 1$ or $k = n$, and you don't want to peek outside of array bounds without due care.

Solution:

```
find_summit(A, start, finish):
    index = floor((start + finish) / 2)
    if (index == start or index == finish):
        return index
    else if (A[index-1] < A[index] and A[index+1] < A[index]):
        return index

    if (A[index+1] < A[index]): // summit must be on the left
        find_summit(A, start, index-1)
    else: // summit must be on the right
        find_summit(A, index + 1, finish)
```

The running time is $O(\log n)$ because the algorithm uses a "divide-and-conquer" strategy, always breaking down the problem in half, and the logarithm describes the maximum amount of splits possible.

(b) Given an integer x, provide an algorithm with running time $O(\log n)$ that checks if x appears in the (valid) array or not. Describe the algorithm either in words or pseudocode and argue about its worst-case running time.

Solution:

To check the existence of x in the array, first find the summit as described previously. If x is not the summit, it must either be to the left or right of it. This splits our original array into two: one ascending and one descending. To find x, simply use binary search on both sides (note that a different variant must be used for either side).

Since finding the summit and binary search both have $O(\log n)$ complexity and they are not nested in one another, the worst-case complexity is also in $O(\log n)$.

Exercise 4.5 Counting function calls in loops (cont'd) (1 point)

For each of the following code snippets, compute the number of calls to f as a function of $n \in \mathbb{N}$. We denote this number by $T(n)$, i.e., $T(n)$ is the number of calls the algorithm makes to f depending on the input n. Then T is a function from $\mathbb{N}$ to $\mathbb{R}^+$. For part (a), provide both the exact number of calls and a maximally simplified asymptotic bound in Θ-notation. For part (b), it is enough to give a maximally simplified asymptotic bound in Θ-notation. You may assume that $n \geq 10$.

(a)

Algorithm 2 Algorithm 2

```
1: i ← 1
2: while i ≤ n do
3:     j ← i
4:     while 2^j ≤ n do
5:         f()
6:         j ← j + 1
7:     end while
8:     i ← i + 1
9: end while
```

Hint: To find the asymptotic bound, it might be helpful to consider n of the form $n = 2^k$.

Solution:

For the inner loop, $2^j \leq n$ was converted to $j \leq \log_2 n$ by applying $\log_2$ on both sides.

$$\sum_{j=i}^{\lfloor \log_2 n \rfloor} 1 = (\lfloor \log_2 n \rfloor - i + 1)$$

Note that $\lfloor \log_2 n \rfloor - i + 1$ is the number of terms in the summation of the form $1 + 1 + ... + 1$. Doing this is necessary as the bounds of the loop are not constant.

Even though the outer loop has n iterations, the inner loop is only executed when the starting condition is fulfilled, that is, when $2^j \leq n$ and since j receives the value of i we have $i \leq \lfloor \log_2 n \rfloor$. This gives us the following summation:

$$\sum_{i=1}^{\lfloor \log_2 n \rfloor} (\lfloor \log_2 n \rfloor - i + 1) = \lfloor \log_2 n \rfloor + \lfloor \log_2 n \rfloor - 1 + ... + 2 + 1$$

Which is simply the sum of the first $\lfloor \log_2 n \rfloor$ numbers, that is

$$\frac{\lfloor \log_2 n \rfloor (\lfloor \log_2 n \rfloor + 1)}{2}$$

which is the exact number of calls to f. Finding the asymptotic growth is made simpler by assuming $n = 2^k$, as this makes the floor function irrelevant (all values of $\log_2 2^k, k \in \mathbb{N}$ are already whole numbers).

$$\frac{\lfloor \log_2 2^k \rfloor (\lfloor \log_2 2^k \rfloor + 1)}{2} = \frac{k(k+1)}{2} = \Theta(k^2)$$

Substituting back $\log_2 n$ gives us the final asymptotic bound

$$\frac{\lfloor \log_2 n \rfloor (\lfloor \log_2 n \rfloor + 1)}{2} = \Theta((\log n)^2)$$

(b)

Algorithm 3 Algorithm 3

```
1: function A(n)
2:     i ← 0
3:     while i < n^2 do
4:         j ← n
5:         while j > 0 do
6:             f()
7:             f()
8:             j ← j − 1
9:         end while
10:        i ← i + 1
11:    end while
12:    k ← ⌊n/2⌋
13:    for l = 0 . . . 3 do
14:        if k > 0 then
15:            A(k)
16:            A(k)
17:        end if
18:    end for
19: end function
```

You may assume that the function $T : \mathbb{N} \to \mathbb{R}^+$ denoting the number of calls of the algorithm to f is increasing.

Hint: To deal with the recursion in the algorithm, you can use the master theorem.

Solution:

We will split this problem into two parts, first calculating the amount of function calls in the nested loop and then analyzing the recursion. As only the asymptotic bound is required, simplifications are made along the way. Also, the inner loop has its bounds rewritten to suit the summation without changing the amount of iterations.

$$\sum_{i=1}^{n^2}\sum_{j=1}^{n} 1 = \sum_{i=1}^{n^2} n = n^2 \cdot n = \Theta(n^3)$$

This is the amount of function calls *without* the subsequent recursions. The for-loop runs 4 times every time A is called and each iteration calls $A(k)$ twice for a total of 8 calls. Since $k = \lfloor \frac{n}{2} \rfloor$ and the floor function is irrelevant for this asymptotic bound we have:

$$T(n) = 8T\left(\frac{n}{2}\right) + n^3$$

Where T is a function that returns the number of calls to f. We can use the master theorem with $a = 8$ and $b = 3$. This gives us case (ii), because $3 = \log_2 8$, which means $T(n) = \Theta(n^3 \log n)$.

(c)

Prove that the function $T : \mathbb{N} \to \mathbb{R}^+$ from the code snippet in part (b) is indeed increasing.

Hint: You can show the following statement by mathematical induction: "For all $n' \in \mathbb{N}$ with $n' \leq n$, we have $T(n'+1) \geq T(n')$."

Solution:

Let's check a base case with $n = 1$. We have $T(1) = 1$ since k will equal 0 (because of the floor function) and all that remains is 1^3.

$$T(2) \geq T(1) \qquad 8T(1) + 2^3 \geq T(1) \qquad 16 \geq 1$$

The base case holds. To easily calculate the floor function, we can consider two cases: n is even, in which case $n = 2m$ and n is odd, meaning $n = 2m+1$ ($m \in \mathbb{N}$ in both cases).

We have:

$$\begin{aligned} T(2m+1) &= 8T\left(\left\lfloor \frac{2m+1}{2} \right\rfloor\right) + (2m+1)^3 \\ &= 8T(m) + (2m+1)^3 \end{aligned}$$

and

$$\begin{aligned} T(2m) &= 8T\left(\left\lfloor \frac{2m}{2} \right\rfloor\right) + (2m)^3 \\ &= 8T(m) + (2m)^3 \end{aligned}$$

Since $(2m+1)^3 > (2m)^3$, we have:

$$T(2m+1) = 8T(m) + (2m+1)^3 > 8T(m) + (2m)^3 = T(2m)$$

The inequality holds. Now for the case where n is odd:

$$\begin{aligned} T(2m+2) &= 8T\left(\left\lfloor \frac{2m+2}{2} \right\rfloor\right) + (2m+2)^3 \\ &= 8T(m+1) + (2m+2)^3 \end{aligned}$$

and

$$\begin{aligned} T(2m+1) &= 8T\left(\left\lfloor \frac{2m+1}{2} \right\rfloor\right) + (2m+1)^3 \\ &= 8T(m) + (2m+1)^3 \end{aligned}$$

Since $n+1 = 2m+1$, we have $m = \frac{n-1}{2} < n$. Let $m = n'$ as in the induction hypothesis. We then have $T(m+1) \geq T(m)$, which implies $8T(m+1) \geq 8T(m)$. Knowing this and the fact $(2m+2)^3 \geq (2m+1)^3$, we can see that the inequality is correct:

$$8T(m+1) + (2m+2)^3 \geq 8T(m) + (2m+2)^3$$

and since both cases hold, it has been proven by the principle of mathematical induction that the function is increasing.