Algorithms and Data Structures: Homework #6

Due on March 16, 2020 at 23:00 PM

Henri Sota

Problem 6.1

a) *Bubble Sort* is a sorting algorithm that works by repeatedly iterating through the list to be sorted, comparing each pair of adjacent items, and swapping them if they are in the wrong order. This is repeated until no swaps are needed, which indicates that the list is sorted. Write down the *Bubble Sort* algorithm in pseudocode including comments to explain the steps and/or actions.

Algorithm 1 Bubble Sort$[A]$

```
// Set swapped to True to enter while loop
swapped = true
// Iterate indefinitely while a swap is available
while true do
  // Set swapped to False as there is no swap yet
  swapped = false
  // Iterate through the array from i to length of array
  for i = 1 to A.length −1 do
    // Check if element in index i is bigger than element in index i + 1
    if A[i] > A[i + 1] then
      // Swap elements in indices i and i + 1
      Swap(A[i], A[i + 1])
      // Set swapped to True
      swapped = true
    end if
  end for
end while
```

```python
def bubbleSort(A):
    # Set swapped to True to enter while loop
    swapped = True
    # Iterate infinitely while a swap is available
    while (swapped):
        # Set swapped to False as there is no swap yet
        swapped = False
        # Iterate through the array from i to length of array
        for i in range(len(A) - 1):
            # Check if element in index i is bigger than element in index i + 1
            if (A[i] > A[i + 1]):
                # Swap elements in indices i and i + 1
                A[i], A[i + 1] = A[i + 1], A[i]
                # Set swapped to True
                swapped = True
    return A
```

Listing 1: Bubble Sort in Python

b) Determine and prove the asymptotic worst-case, average-case, and best-case time complexity of *Bubble Sort*.

Worst Case: Bubble Sort's worst case performance happens when the array to be sorted is reverse sorted to the way Bubble Sort sorts the array. The while loop would have to run the code inside its body for $n-1$ iterations as $n-1$ elements would have to be placed in their correct position with the n-the element would be automatically placed in its position by the other swaps. The inner loop would be evaluated n times for the first iteration of the while loop, as we need to check the first element of

the array with all the other ones, $n-1$ times for the second iteration etc. The recurrence for Bubble Sort's worst case is:

$$\sum_{i=0}^{n-1} n-i-1 = n^2 - n - \sum_{i=0}^{n-1} i = n^2 - n - \frac{n(n-1)}{2} = \frac{2n^2-2n-n^2-n}{2} = \frac{n^2}{2} - \frac{n}{2}$$

$$\begin{aligned} T(n) &= O(n-1) + n \\ &= O(n^2) \end{aligned}$$

Average Case: Bubble Sort's average case performance happens when the elements are sorted in a way that there is a need for only half of the maximum swaps. As there is only a constant factor difference between this case and worst case, they both belong to the same class of O-notation. The solution to the recurrence for Bubble Sort's average case is:

$$T(n) = O(n^2)$$

Best Case: Bubble Sort's best case performance happens when the array to be sorted is already sorted in the way Bubble Sort sorts the array. The while loop would have to run only once as after the first iteration, because swapped would still hold the value False (if clause never evaluates to True as $A[i] \leq A[i+1]$, $\forall i \in [0, A.[\text{length}-1]]$) and therefore the loop would finish iterating. The solution to the recurrence for Bubble Sort's best case is:

$$T(n) = O(n)$$

c) Stable sorting algorithms maintain the relative order of records with equal keys (i.e., values). Thus, a sorting algorithm is **stable** if whenever there are two records R and S with the same key and with R appearing before S in the original list, R will appear before S in the sorted list. Which of the sorting algorithms *Insertion Sort*, *Merge Sort*, *Heap Sort*, and *Bubble Sort* are stable? Explain your answers.

- **Insertion Sort** - Stable
  Insertion Sort's ineer loop stops iterating if it finds an element that is smaller or equal to the key. It means that it won't perform a swap with between the key and the element at position found when both elements are equal to each other. Therefore, Insertion Sort maintains stable property when sorting elements.
- **Merge Sort** - Stable
  Merge Sort's Merge procedure is considered to be stable. When merging the two sorted subarrays, the elements of the left subarray which were ordered before the elements of the right subarray will be placed first in the merged array if they are smaller or equal to the element they are being compared to in the right subarray. Therefore, Merge Sort maintains stable property when sorting elements.
- **Heap Sort** - Unstable
  Heap Sort's Build Max Heap procedure is considered to be unstable. While creating the Max Heap, the former positions of the elements in the to-be sorted array are lost. In the next procedure of iteratively popping the biggest element of the heap and placing it in its sorted position, the ordering of the elements in the heap will change again yielding an unstable sort. There is also a case when the initial positions of the elements was such that Heap Sort would act as a stable sort, but that is rare. Therefore, Heap Sort does not maintain stable property when sorting elements.
- **Bubble Sort** - Stable
  Bubble Sort won't change the order of the elements with the same key. This is due to the if clause that only swaps the elements if the element in index i is strictly greater than the element in index $i+1$. Two elements with the same key will remain in the same positional relation with each other. Therefore, Bubble Sort maintains stable property when sorting elements.

d) A sorting algorithm is **adaptive**, if it takes advantage of existing order in its input. Thus, it benefits from the pre-sortedness in the input sequence and sorts faster. Which of the sorting algorithms *Insertion Sort*, *Merge Sort*, *Heap Sort*, and *Bubble Sort* are adaptive? Explain your answers.

- **Insertion Sort** - Adaptive
  Insertion Sort's inner loop stops iterating if it finds an element that is smaller or equal than the key. The loop invariant holds the property that the subarray $[0 \cdots j-1]$ is already sorted. The inner loop is used to find the position where to place the key in the already sorted subarray. In the best case, the inner loop would evaluate to false every time the outer loop would run. Insertion Sort's time complexity in the best and worst case is:
  - Best Case Complexity: $T(n) = O(n)$
  - Worst Case Complexity: $T(n) = O(n^2)$
- **Merge Sort** - Not Adaptive
  Merge Sort's recursive calls and comparisons are called $\log(n)$ and $n\log(n)$ times regardless of the pre-sortedness of the array. Merge Sort will divide the array into subarrays recursively untill arriving to the base case of 1. The Merge procedure combines two subarrays into one by doing $n-1$ comparisons, where n is the number of elements in both subarrays. Merge Sort's time complexity in the best and worst case is:
  - Best Case Complexity: $T(n) = O(n \log n)$
  - Worst Case Complexity: $T(n) = O(n \log n)$
- **Heap Sort** - Not Adaptive
  Heap Sort's Build Max Heap procedure gets rid of any sorted relationship of the elements of the initial array in the general case. Due to the data structure used and the last procedure of iteratively popping off the biggest element, Heap Sort does not profit from the pre-sortedness of initial array. The swaps inside of Max Heapify procedure might vary depending on the frequency and distribution of the elements (best case of $O(n)$ with equal keys), but that still doesn't affect the order of complexity of Heap Sort, because running sort on an array filled with the same elements is not useful. Heap Sort's time complexity in the best and worst case is:
  - Best Case Complexity: $T(n) = O(n \log n)$
  - Worst Case Complexity: $T(n) = O(n \log n)$
- **Bubble Sort** - Adaptive
  Bubble Sort's outer loop stops iterating if value of the swap flag is set to False. If the value of swap is set to false in the beginning of the iteration, this implies that no swaps were needed on the previous iteration (elements were in the desired order between each other). Bubble Sort's time complexity in the best and worst case is:
  - Best Case Complexity: $T(n) = O(n)$
  - Worst Case Complexity: $T(n) = O(n^2)$

Problem 6.2

a) Implement the *Heap Sort* algorithm as presented in the lecture.

Algorithm 2 Left[i]

return $i * 2$

Algorithm 3 Right[i]

return $i * 2 + 1$

Algorithm 4 Max Heapify[A, i]

$l = \text{Left}(i)$
$r = \text{Right}(i)$
if $l \leq A.heapsize$ and $A[l] > A[i]$ **then**
  $largest = l$
else
  $largest = i$
end if
if $r \leq A.heapsize$ and $A[r] > A[largest]$ **then**
  $largest = r$
end if
if $largest \neq i$ **then**
  $Swap(A[i], A[largest])$
  Max Heapify($A, largest$)
end if

Algorithm 5 Build Max Heap[A]

$A.heap\text{-}size = A.length$
for $i = \lfloor A.length/2 \rfloor$ **to** 1 **do**
  Max Heapify(A, i)
end for

Algorithm 6 Heap Sort[A]

Build Max Heap(A)
for $i = A.length$ **to** 2 **do**
  $Swap(A[1], A[i])$
  $A.heap\text{-}size = A.heapsize - 1$
  Max Heapify($A, 1$)
end for

```python
def left(i):
    # Return index of left child
    return i * 2 + 1

def right(i):
    # Return index of right child
    return i * 2 + 2

def maxHeapify(A, i, heapsize):
    l = left(i)
    r = right(i)

    # Check if left child exists and is greater than element
    if ((l < heapsize) and (A[l] > A[i])):
        # Set largest to the index of left child
        largest = l
    else:
        # Set largest to the index of element
        largest = i

    # Check if right child exists and is greater than the largest-th element
    if ((r < heapsize) and (A[r] > A[largest])):
        # Set largest to the index of right child
        largest = r

    # Check if largest and i are different
    if (largest != i):
        # Swap between the largest child and the parent of the child
        A[i], A[largest] = A[largest], A[i]
        # Call maxHeapify on the largest-th element which holds the value of the
        # former parent of largest-th element
        A = maxHeapify(A, largest, heapsize)
    return A

def buildMaxHeap(A, heapsize):
    # Loop from the parent of the heapsize-th element to root
    for i in range((heapsize - 1) // 2, -1, -1):
        # Maintain Max Heap property by calling maxHeapify
        A = maxHeapify(A, i, heapsize)
    return A

def heapSort(A, heapsize):
    # Build Max Heap from the unsorted array
    buildMaxHeap(A, heapsize)
    # Pop biggest element off heap and maintain Max Heap property iteratively
    for i in range(len(A) - 1, 0, -1):
        A[0], A[i] = A[i], A[0]
        heapsize -= 1
        A = maxHeapify(A, 0, heapsize)
    return A
```

Listing 2: Heap Sort in Python

b) Implement a variant of the *Heap Sort* that works as follows: In the first step it also builds a max-heap. In the second step, it also proceeds as the *Heap Sort* does, but instead of calling $MAX-HEAPIFY$, it always floats the new root all the way down to a leaf level. Then, it checks whether that was actually correct and if not fixes the max-heap by moving the element up again. This strategy makes sense when considering that the element that was swapped to become the new root is typically small and thus would float down to a leaf level in most cases. Hence, one would save the additional tests when floating down the element. And, the fixing step (moving the element upwards again) would be a rare

case.

Algorithm 7 Sift Up$[A, i]$

if $A[\text{parent}(i)] < A[i]$ **then**
  $Swap(A[l\text{parent}(i)], A[i])$
  Sift Up$(A, \text{parent}(i))$
end if

Algorithm 8 Sift Down$[A, i]$

$l = \text{Left}(i)$
$r = \text{Right}(i)$
if $l \geq$ heapsize **then**
  return i
else if $r \geq$ heapsize **then**
  $Swap(A[l], A[i])$
  return l
else
  if $A[l] > A[r]$ **then**
    $Swap(A[l], A[i])$
    largest $= l$
  else
    $Swap(A[r], A[i])$
    largest $= r$
  end if
  return Sift Down(A, i)
end if

Algorithm 9 Heap Sort Variant$[A]$

Build Max Heap(A)
for $i = A.length$ **to** 2 **do**
  $Swap(A[1], A[i])$
  $A.heap\text{-}size = A.heapsize - 1$
  Sift Down$(A, 1)$
end for

```
def parent(i):
    # Return index of parent
    return (i - 1) // 2

def left(i):
    # Return index of left child
    return i * 2 + 1

def right(i):
    # Return index of right child
    return i * 2 + 2

def maxHeapify(A, i, heapsize):
```

```
    l = left(i)
    r = right(i)

    # Check if left child exists and is greater than element
    if ((l < heapsize) and (A[l] > A[i])):
        # Set largest to the index of left child
        largest = l
    else:
        # Set largest to the index of element
        largest = i

    # Check if right child exists and is greater than the largest-th element
    if ((r < heapsize) and (A[r] > A[largest])):
        # Set largest to the index of right child
        largest = r

    # Check if largest and i are different
    if (largest != i):
        # Swap between the largest child and the parent of the child
        A[i], A[largest] = A[largest], A[i]
        # Call maxHeapify on the largest-th element which holds the value of the
        # former parent of largest-th element
        A = maxHeapify(A, largest, heapsize)
    return A

def buildMaxHeap(A, heapsize):
    # Loop from the parent of the heapsize-th element to root
    for i in range((heapsize - 1) // 2, -1, -1):
        # Maintain Max Heap property by calling maxHeapify
        A = maxHeapify(A, i, heapsize)
    return A

def siftUp(A, i):
    # Move element up while the parent is smaller
    if (A[parent(i)] < A[i]):
        A[i], A[parent(i)] = A[parent(i)], A[i]
        siftUp(A, parent(i))

def siftDown(A, i, heapsize):
    # Float new root down the tree to a leaf level
    l = left(i)
    r = right(i)
    # Check if there are no leaves connected to the root
    if l >= heapsize:
        return i
    # Check if there is only one leaf connected to the root
    elif r >= heapsize:
        A[i], A[l] = A[l], A[i]
        return l
    else:
        # Compare only the leaves to find the biggest path to follow down
        if A[l] > A[r]:
            A[i], A[l] = A[l], A[i]
            largest = l
        else:
            A[i], A[r] = A[r], A[i]
            largest = r
        # Recursively call itself to move one level further down
        return siftDown(A, largest, heapsize)

def heapSortVariant(A, heapsize):
    # Build Max Heap from the unsorted array
```

```
    buildMaxHeap(A, heapsize)
    # Pop biggest element off heap and maintain Max Heap property iteratively
    for i in range(len(A) - 1, 0, -1):
        A[0], A[i] = A[i], A[0]
        heapsize -= 1
        # Call siftDown and siftUp in place of maxHeapify
        position = siftDown(A, 0, heapsize)
        siftUp(A, position)
    return A
```

Listing 3: Heap Sort with Bottom Up Variant in Python

Bottom Up Variant Implementation using Sift Down and Sift Up was based on the variant algorithm of Heap Sort here.