Data Structures and Introduction to Algorithms - 20407

Maman 12

Yehonatan Simian
206584021

Question 1 (25%)

Section 1

What is the number of comparisons performed during a `BUILD-MAX-HEAP` procedure for an input array of size n whose elements are all identical?

Solution Let A an array of size n whose elements are all identical. A's elements are identical, hence $\forall_{1 \leq i,j \leq n} : A[i] \leq A[j]$, therefore the two comparisons on steps 3 and 6 of `MAX-HEAPIFY` will always return `false`, hence the comparison on step 8 will return `false` as well, resulting in exactly three comparisons per call of `MAX-HEAPIFY`.

`MAX-HEAPIFY` is being called $\lfloor n/2 \rfloor$ times according to step two of `BUILD-MAX-HEAP`, resulting in a total number $3 \cdot \lfloor n/2 \rfloor$ comparisons performed during the `BUILD-MAX-HEAP` procedure. ■

Section 2

Given a min-heap whose elements in each depth are identical, write a pseudo-code procedure that receives such a heap as well as another element z as input, and returns the index of an instance of z in the heap or -1 if it does not exist in the heap.

```
def foo(A,z):
    index = 1
    while index < heap-size(A):
        if A[index-1] == z:
            return index
        index *= 2
    return -1
```

Brief explanation: in each iteration of the loop, we compare z to one node (representing all node in a certain depth), proceeding with the left son of that node in case the comparison failed. Since `index` double itself each iteration (i.e. proceeding to node's son), the number of iterations is logarithmic with the size of the heap, resulting in the requested logarithmic runtime of the procedure. ■

Section 3

Given an array of size n that contains a max-heap of an unknown size x in its first x elements (and the rest of the elements are ∞), you must determine x in a logarithmic time in x.

Solution Let's divide the procedure to determine x into two parts:

1. Determine the depth of the heap. This part has a logarithmic runtime as for all needed is doubling an index (starting from 1) until we reach ∞.

2. Use binary search to find the index of the first occurrence of ∞.

Question 1 continued on next page. . .

Using these two logarithmic sub-routines we can determine x easily, being the predecessor of the index found in part 2 (which has logarithmic runtime by the definition of binary search). ■

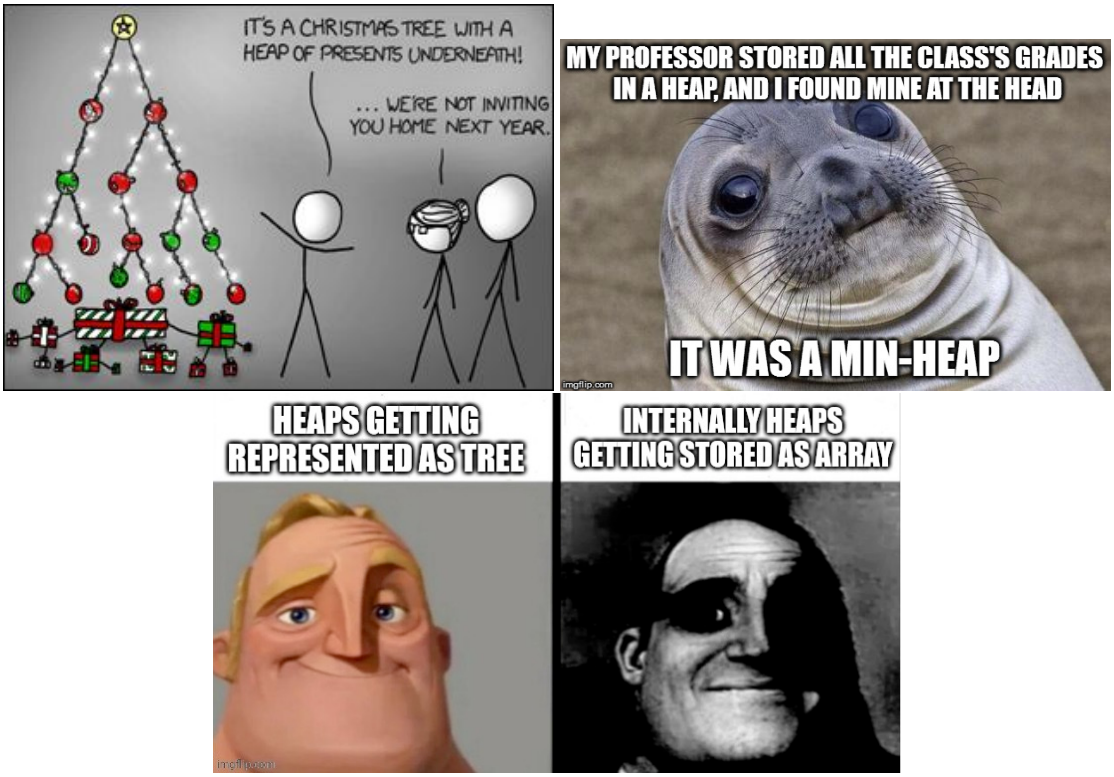

Figure 1: Heap Memes

Question 2 (25%)

Section 1

Prove that every comparison-based sorting must perform at least 7 comparisons (worst-case) on an array of size 5.

Proof As mentioned in Theorem 8.1, the depth h of the decision tree of any comparison-based sort of A is bounded by $h \geq \log n$. Since $n = 5$ in our case, $h \geq \log 5 \approx 6.9$, hence the number of comparisons (which is the longest length from the root of the decision tree to a leaf in that tree) is at least 7. ■

Section 2

How many comparisons are performed on each of the sorting algorithms we've learned on an array of size 5 in the best case and in the worst case?

Solution

- **Insertion Sor**t:
  - Best-case: the array is already sorted, hence step 5 of the `INSERTION-SORT` algorithm will always return false, which results in $n - 1 = 4$ comparisons (due to the loop on step 1).
  - Worst-case: the array is sorted in reverse order, hence step 5 of the algorithm will run i times each iteration of step 1, resulting in $\sum_{i=1}^{n-1} i = 10$ comparisons.
- **Merge Sort**: in both best-case and worst-case, exactly 8 comparisons will be performed. The reason is that the `MERGE` procedure always makes the same number of comparisons when merging the sub-arrays (and no others comparisons are done). More specifically, the "divide" phase of the algorithm will:
  - Split 5 elements into 3 and 2.
  - Split 3 elements into 2 and 1.
  - Split 2 elements into 1 and 1.
  - Split 2 elements into 1 and 1.

  And then the "conquer" phase of the algorithm will:
  - Merge two single-element arrays (1 comparison).
  - Merge two single-element arrays (1 comparison).
  - Merge a single-element array with a double-element array (2 comparisons).

Question 2 continued on next page. . .

- Merge a double-element array with a triple-element array (4 comparisons).

Resulting with exactly 8 comparisons as promised.

- **Heap Sort**:
  - Best-case: in this scenario the array is already sorted, resulting in exactly 3 comparisons per call of `MAX-HEAPIFY` (according to the explanation on Question 1). `MAX-HEAPIFY` is being called $\lfloor 5/2 \rfloor$ times during `BUILD-MAX-HEAP`, and also $5 - 1$ times during step 5 of `HEAPSORT`, resulting in $3 \cdot (2 + 4) = 18$ comparisons.
  - Worst-case: in this scenario, step 10 of `MAX-HEAPIFY` will be called recursively, adding 5 more exchanges in the tree (in order to sort it properly), resulting in a total of $18+5\cdot 3 = 33$ comparisons during `HEAPSORT`.
- **Quick Sort**:
  - Best-case: in this case, the partitioning is balanced as possible, resulting in 4 comparisons for the first partitioning (comparing each of the first 4 elements with the pivot) plus one comparison per each of the two two-elements partitions, hence the total number of comparisons in the best-case scenario is $4 + 1 + 1 = 6$.
  - Worst-case: similar to Exercise 7.2-2 in the book (a scenario where all elements are identical, hence the condition on step 4 of `PARTITION` will always be evaluated to `true`), the number of comparisons is $\sum_{i=2}^{5} i = 14$.

Section 3

Write a procedure that sorts an array of size 5 using 7 comparisons at most.

```
SORT5(A)
    if A[0] > A[1]
        exchange(A[0], A[1])
    if A[2] > A[3]
        exchange(A[2], A[3])
    if A[0] > A[2]
        exchange(A[0], A[2])
        exchange(A[1], A[3])
    if A[1] > A[4]
        exchange(A[1], A[4])
    if A[1] > A[2]
        exchange(A[1], A[2])
    if A[3] > A[4]
        exchange(A[3], A[4])
    if A[2] > A[3]
        exchange(A[2], A[3])
```

NOTE that's basically a pure decision tree approach. ■

Question 3 (25%)

Prove or disprove:

Section 1

It is possible to sort a min-heap in linear time.

Solution Assume such sorting is possible. Let's call the sorting routine `SORT-MIN-HEAP`. We can use it to implement `MINHEAPSORT`:

```
MINHEAPSORT(A)
    BUILD-MIN-HEAP(A)
    SORT-MIN-HEAP(A)
```

The routine `BUILD-MIN-HEAP` is suggested on page 113 in the book, as well as the linearity of its runtime. Both routines have linear runtime, thus the runtime of `MINHEAPSORT` is also linear, resulting in a linear comparison-based sorting algorithm which cannot exist according to Theorem 8.1 in the book. We reached a contradiction, hence it is **not** possible to sort a min-heap in linear time. ■

Section 2

It is possible to find and sort the smallest $n/\log n$ elements in an array in linear time.

Solution Here's a detailed breakdown of the algorithm:

1. Find the $n/\log n$-th element in the array using `SELECT` ($O(n)$).
2. Isolate the smallest $n/\log n$ elements using `PARTITION` ($O(n)$).
3. Sort the required partition using `HEAPSORT`. The partition has a size of $n/\log n$, resulting in runtime of $n/\log n \cdot \log(n/\log n) = n/\log n \cdot \log n - n/\log n \cdot \log\log n = O(n)$.

Each of the steps takes linear runtime, eventually resulting in the required result, thus it **is** possible to find and sort the smallest $n/\log n$ elements in an array in linear time. ■

Question 3 continued on next page. . .

Section 3

It is possible to sort an array of size n in which each element is bounded by $2n\sqrt{n}$ in linear time.

Solution Ngl, my solution is a bit controversial because it was *not* explicitly given that the input distributes uniformly... but I'll assume it does distributes uniformly (which makes sense due to the input being random) because I have a ~~sexy~~ nice solution:

1. Divide each element by $2n\sqrt{n} + 1$.
2. Perform Bucket Sort on the array.

Each of the steps is linear, and the first step ensures that all elements in the resulted array's are in the range $[0, 1)$, which is a requirement of Bucket Sort. ■

NOTE If the solution is invalid due to the usage of Bucket Sort when it was not explicitly given that the input distributes uniformly, then is it ***not*** *possible to sort an array of size n in which each element is bounded by* $2n\sqrt{n}$ *in linear time due to the lower-bound of comparison-based algorithms and the requirement of counting sort that the range of the input is* $O(n)$.

Section 4

Given an array of size n that stores the integers from 0 to n except one. It is possible to find the missing number in linear time.

Solution What a classical and wonderful riddle of fifth grade! Oh, how marvelous and fantastic question have you brought me upon this magnificent, glorious day. It is my pleasure to introduce you the tremendous solution of this beautiful master-piece of a mystery... okay I'll stop the nonsense here. You just sum-up the entire array and subtract it from $\sum_{i=0}^{n} = n(n+1)/2$ to get the result. ■

Question 4 (25%)

Let A an array of size n where each element is either 0, 1 or 2.

Section 1

This is a recurrence of Question 2 Section 2:

- In the best-case, A is already sorted, resulting in $n - 1$ comparisons.
- In the worst-case, A is sorted descendently, resulting in $n(n-1)/2$ comparisons.

NOTE the restriction of the elements being 0, 1 or 2 does not affect the number of comparisons.

Section 2

Sort A by two calls to `PARTITION(A,p,r)`. Before each call you may traverse A only once in order to determine p and r.

Solution The procedure `PARTITION` takes a pivot element and re-arranges the array such that all elements on its left are lesser or equal to the pivot, and all elements on its right are greater than it. For that reason, two iterations of `PARTITION` are just sufficient to sort the array: first we scan the array in order to find a "0" element and use it as a pivot with `PARTITION(A,1,p)` where `p` is the location of "0" (that way all zeros go to the beginning of the array); then we scan the array to find a "1" element and use it as a pivot with `PARTITION(A,q,t)` where `q` is the result of the first `PARTITION` call and `t` is the location of "1" (that way all ones go to the beginning of the sub-array, and all that's left is twos at the end of the array). ■

NOTE If an element has not been found on either scans of the array, there's no need to call `PARTITION` after that scan. E.g., if both scans failed, then the array contains only "2" elements and is trivially sorted.

Section 3

Use Counting Sort. What is the size of B and C array in which you must use?

Solution B should be the same size as A, i.e. n, and C should be of size 3 in order to store the amounts of 0, 1 and 2 in A. ■

Question 4 continued on next page. . .

Section 4

Which of the sorting algorithms above are stable?

Solution Insertion Sort and Counting Sort are stable by definition. Quick Sort is not stable even with the restriction of the elements being either 0, 1 or 2, due to the exchanges on lines 6 and 7 of the `PARTITION` procedure. ■

Figure 2: Muslim Sonic Meme