CS 581 Homework 4 Solution

02/12/2018

Problems (Due on 02/08/2018)

Problem 1. A previous CS581 student (let's call him Jerome) took this class and proposed the following sorting algorithm:

```
JSORT(LIST)
if LIST.length=0,1 then return LIST
if LIST.length=2 then compare items, swap if necessary and return LIST
JSORT the leftmost two-thirds of LIST
JSORT the rightmost two-thirds of LIST
JSORT the leftmost two-thirds of LIST
return LIST
```

a) Argue convincingly either that JSORT does or that it does not work.
b) In any case, determine JSORT's time complexity.

Possible Answer . a) This algorithm works when we always divide the array into three parts of equal lengths.

We could raise n to the closest 3^k where $k \in \mathbb{N}$ by padding some extra keys. The extra key could be some number that is less than any number in the list. After the whole array is sorted, we remove the extra keys. Note that both adding and removing extra keys can be done in constant time. After the padding we don't need to worry about ceiling or flooring anymore. A possible argument could be as follows:

Assume a non-decreasing order.

- Each 1/3 subgroup is sorted itself, so we only consider the possible inversions across the groups.
- The first 1/3 and the second 1/3 are sorted most recently (the last 2/3 JSORT operation), so no inversion can happen between them.
- After the first 2/3 JSORT and the second 2/3 JSORT, every element in the first 2/3 group should be no greater than the last 1/3 group. Although the first 2/3 group is sorted later again to guarantee internal sortedness, every element in the first 2/3 group should still be no greater than the last 1/3 group. So no inversion can happen between the second 1/3 group and the third 1/3 group as well

Therefore, we have

- Each 1/3 group is sorted
- Each element in the first 1/3 group is no greater than the second 1/3s group
- Each element in the second 1/3 group is no greater than the third 1/3s group

Therefore, the array should be sorted as a whole.

`JSORT` can be considered some kind of variant of bubble sort, but with each element to bubble up being a portion of the array. Using the similar idea, we could have another algorithm like this:

```
JSORT(LIST)
if LIST.length=0,1 then return LIST
if LIST.length=2 then compare items, swap if necessary and return LIST
JSORT the first 2/4 of LIST
JSORT the second 1/4 and the third 1/4 of LIST
JSORT the last 2/4 of LIST
JSORT the first 2/4 of LIST
JSORT the second 1/4 and the third 1/4 of LIST
JSORT the first 2/4 of LIST
return LIST
```

If we keep dividing the subarray into smaller pieces, we will get an original bubble sort eventually (base case for each subarray), with a n^2 time complexity.

b) JSORT's recurrence conforms to master theorem, which is $n^{log_{1.5}3}$. Note that raising n to a 3^k does not affect this asymptotic complexity, since that is a linear transformation.

Problem 2.
Prove that the running time of building a heap is $O(n)$. You can assume a MAX-heap.

Possible Answer . The idea is to sink in a bottom-up fashion. There are more lower nodes but they also travel a shorter distance. You can refer to the textbook for a detailed proof.

When Heapify is called, run-time depends on how many times the element could be moved downwards. The worst case is to move the node to the bottom. With a height of h (starting at 0 for the root), there are 2^h leaf nodes. At the level above that there are 2^{h-1} nodes, and there might be 1 shift downward for each node. At level j, there are 2^{h-j} nodes that all might shift j levels. So,

$$T(n) = \Sigma_{j=0}^{h}(j2^{h-j}) = (2^h)\Sigma_{j=0}^{h}(\frac{j}{2^j}) \tag{1}$$

For infinite geometric series where $x < 1$

$$\Sigma_{j=0}^{\infty}(x^j) = \frac{1}{1-x} \tag{2}$$

Take derivative of both sides

$$\Sigma_{j=0}^{\infty}(jx^{j-1}) = \frac{1}{(1-x)^2} \Rightarrow \Sigma_{j=0}^{\infty}(jx^{j}) = \frac{x}{(1-x)^2} \tag{3}$$

When $x = \frac{1}{2}$,

$$\Rightarrow \Sigma_{j=0}^{\infty}(j\frac{1}{2^j}) = \frac{\frac{1}{2}}{(\frac{1}{2})^2} = 2 \Rightarrow T(n) = 2^h \times 2 = 2^{h+1} \tag{4}$$

Since $n = 2^{h+1} - 1, T(n) \in \Theta(n)$, so $T(n) \in \mathcal{O}(n)$.

Problem 3.
Determine if array {27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0} is a MAX-heap. If not, illustrate the operation of Max-heapify by sinking or floating down nodes. This assumes the storage scheme described in the textbook.

Possible Answer . Note that it says assuming the storage scheme described in the textbook, meaning the two children of node i would be stored at $2i$ and $2i + 1$. And the index of the first element is 1 not 0.

Max-heapify:
Sink node 3: 27 17 10 16 13 3 1 5 7 12 4 8 9 0
Sink node 3 again: 27 17 10 16 13 9 1 5 7 12 4 8 3 0
You can draw a graph as well. I'll omit that here.

Problem 4. You are given the job of partitioning $2n$ players into two teams of n players each. Each player has a numerical rating that measures how good he/she is at the game. Newt seeks to divide the players as unfairly as possible, so as to create the biggest possible talent imbalance between team A and team B. Show how the job can be done in $O(n \log n)$ time.

Possible Answer . Sort the players by their talent rating using any $\Theta(n \log n)$ sort. The greatest disparity would be using the first and second half of this sorted array. You can even do a linear selection to find the median first and then do one quicksort-like partition. That would be linear time.

Problem 5. Suppose an array A consists of n elements, each of which is red, white, or blue. We seek to sort the elements so that all the reds come before all the whites, which come before all the blues. The only operation permitted on the keys are

- Examine(A, i): report the color of the ith element of A.

- Swap(A, i, j): swap the ith element of A with the jth element.

Find, describe and analyze a linear time algorithm for this red-white-blue sorting.

Possible Answer . The Dutch National Flag problem can be solved in linear time. This requires a pass through the array, putting one color on one side and another color on the other side of the array.

Algorithm 1 Dutch National Flag problem

```
Require: Array A, r = 0, b = A.size
  while r ≤ b do
    if A[i] is RED then
      Swap(A[i], A[r])
      r = r + 1
      i = i + 1
    else if A[i] is BLUE then
      Swap(A[i], A[b])
      b = b - 1
    else
      i = i + 1
    end if
  end while
```

It is clear from the pseudo-code that the algorithm runs in linear time.

Another solution is to do two passes and each pass is a two-way partition. It'll still be linear time.