CMSC 351 Fall 2023 Homework 6 Solutions

Due Wednesday Oct 20, 2023 by 11:59pm on Gradescope.

Directions:

- Homework must be done on printouts of these sheets and then scanned properly, or via latex, or by downloading, writing on the PDF, and uploading.
- Do not use your own blank paper!
- The reason for this is that gradescope will be following this template to locate the answers to the problems so if your answers are organized differently they will not be recognized.
- Tagging is automatic, do not manually tag.

1. Suppose we have a 1-indexed list A which represents a complete binary tree. We call `maxheapify` on the nodes with indices $i = 5, 4, 3, 2, 1$ in that order. Show the state of the list after each call. Note that each call includes the recursive calls it makes. [10 pts]

Index	1	2	3	4	5	6	7	8	9	10
Original A	10	3	6	4	1	8	9	5	7	1
After `maxheapify(5)`	10	3	6	4	2	8	9	5	7	1
After `maxheapify(4)`	10	3	6	7	2	8	9	5	4	1
After `maxheapify(3)`	10	3	9	7	2	8	6	5	4	1
After `maxheapify(2)`	10	7	9	5	2	8	6	3	4	1
After `maxheapify(1)`	10	7	9	5	2	8	6	3	4	1

2. One of the problems with implementing Heap Sort in real life is its unfriendliness with the cache. In real computers, it is much faster to access indices that are close to those you have just accessed than it is to access indices that are farther away.

   Suppose we have a complete binary tree of size $n = 2^k - 1$ for some $k \geq 3$. Assume also that:

   - C is a fixed positive constant.
   - Swapping the element in index i with any of the indices in the range $[i-4, i+4]$ inclusive (a *cache hit*) takes C time.
   - Swapping the element in index i with any other index outside that range (a *cache miss*) takes $4C$ time.
   - Only swaps take any time at all, and nothing else takes time.

   (a) Compute the worst-case runtime $T(n) = ...$ of completely `maxheapify`-ing the root of this heap. This should be an actual runtime and not Θ, $\mathcal{O}$, or Ω. Write your answer as a function of n. [5 pts]

Worst-case runtime:	$T(n) = 2C + 4C(\lg(n+1) - 3)$

   (b) Use your answer in part (a) to compute the worst-case runtime $T(n) = ...$ of the very first iteration of `heapsort`'s `for` loop on this heap. This should be an actual runtime and not Θ, $\mathcal{O}$, or Ω. Write your answer as a function of n. [5 pts]

Worst-case runtime:	$T(n) = 6C + 4C(\lg(n+1) - 3)$

   (c) Explain your answer to (a). [10 pts]

   Solution:

   Suppose $k = 4$ so $n = 15$. The worst case scenario implies we should make swaps at every single level. From level 1 to 2 and level 2 to 3, the maximum difference in indices is 4, so we have $2c$. From level 3 to 4 and all future swaps, the largest difference in indices is greater than 4, so we have $4c$ for the rest of the swaps. Note that we have $\lg(n+1) - 3$ remaining levels in the binary tree, so we obtain

$$T(n) = 2c + 4c(\lg(n+1) - 3)$$

3. Comparison of Sorting Algorithms! [6 pts]

Suppose we have the list $A = [5, 3, 1, 2, 4]$. How many swaps does each of the following sorting algorithms require:

Algorithm	Number of Swaps
Bubble Sort	6
Selection Sort	4
Heap Sort	8

4. Consider the following pseudocode for `partition`: [10 pts]

```
function partition(A)
    n = len(A)-1
    pivotvalue = A[n]
    t = 0
    for i = 0 to n-1 inclusive
        if A[i] <= pivotvalue
            A[t] <-> A[i]
            t = t + 1
        end if
    end if
    A[t] <-> A[R]
    return t
end function
```

Suppose we call `partition(A)` on `A=[10,1,9,2,8,5)`. Show the state of the list at the indicated instances.

Initial A	10	1	9	2	8	5
After $i = 0$ ends	10	1	9	2	8	5
After $i = 1$ ends	1	10	9	2	8	5
After $i = 2$ ends	1	10	9	2	8	5
After $i = 3$ ends	1	2	9	10	8	5
After $i = 4$ ends	1	2	9	10	8	5
After final swap	1	2	5	10	8	9

5. After running partition on an array, the array is as follows:

Index	0	1	2	3	4	5	6	7	8	9
Value	33	45	12	60	63	48	72	89	75	76

(a) Answer the questions below: [4 pts]

What element is telling us that 45 was NOT the pivot?	12
What element is telling us that 60 was NOT the pivot?	48

(b) What element in the array was the pivot when partition was run? [2 pts]

Only possible element that was the pivot when partition was run?	72

Explain your answer above. [4 pts]

Solution:

The partition function chooses the last element in an array to be a "pivot value" and it partitions the array such that all elements to the left of the pivot value in the final array are less than the pivot value, and all elements to the right of the pivot value are greater than the pivot value. In the above example, 72 is the only element that has all numbers to its left smaller than it and all numbers to its right greater than it.

6. In the previous question, we can tell what the pivot (originally the last element in the array) was by looking at the array. Sometimes we cannot. Assuming that the array is made up of the elements 25, 42, 18, and 37, not necessarily in this order.
Come up with the state of the array after partition is run once so that we cannot tell for sure what the pivot was. [5 pts]

Index	0	1	2	3
Value	18	25	37	42

7. We are running partition once on the following array (using the last element as the pivot): [5 pts]

$$[65, 50, 78, 10, 99, 46, 15, 90, 60]$$

For some reason (power failure, for example), partition has crashed (and stopped) at the end of an i iteration (guaranteed); the array now looks like this:

$$[50, 10, 46, 65, 99, 78, 15, 90, 60]$$

At the end of what possible i iteration has partition crashed? Note: The first iteration is for $i = 0$. If you think it crashed after the first iteration, then your answer should be 0.

At the end of what possible i iteration has partition crashed?	5

Put scratch work below. Scratch work is not graded but note that you should know how to explain your answer because you may be expected to do this on an exam.

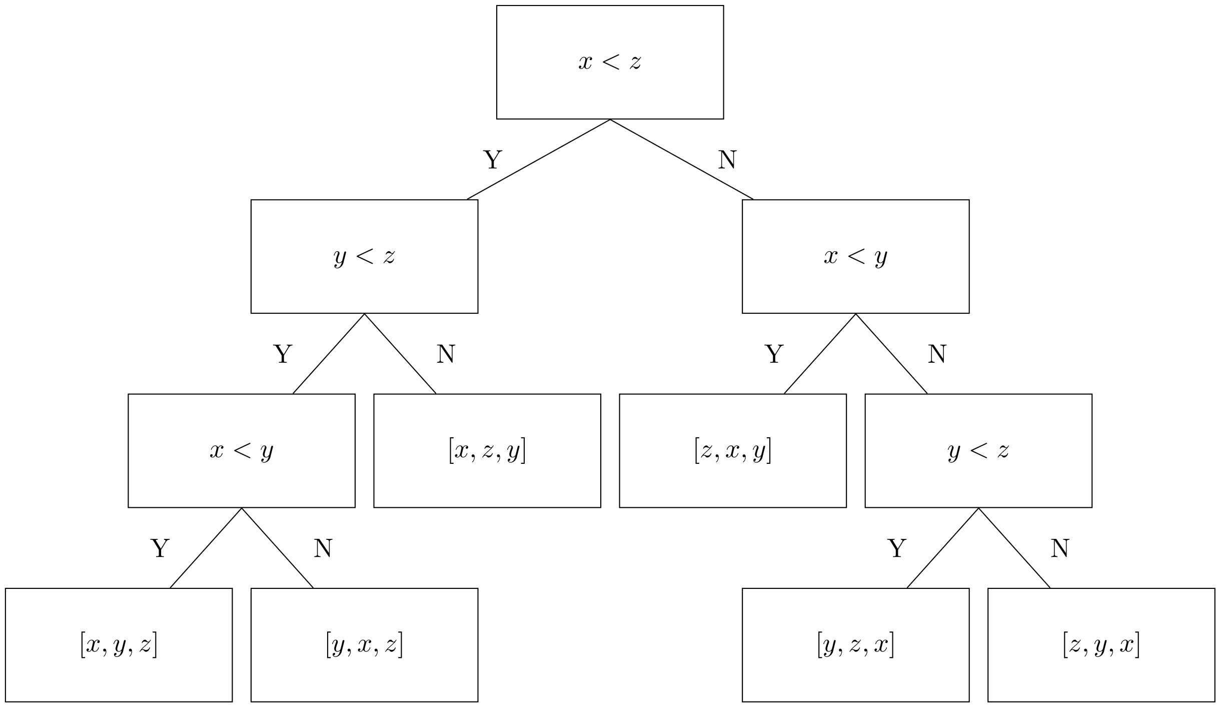
8. We are running partition once on an array of 5 UNIQUE elements. How many possible arrangements of the array values are there so that partition will split the array into 1 element on the left and 3 elements on the right (the pivot, originally the last element in the array, ending up exactly at index 1 of the array).
Arrangement examples: If the array is made up of the elements 10, 20, 30, 40, 50, possible arrangements include [10,30,50,40,20], [10,50,30,40,20], ...

(a) How many possible arrangements are there? [5 pts]

Number of Possible Arrangements is:	4! = 24

(b) Explain your answer above. [5 pts]
Given that the partition function will split the array into 1 element on the left and 3 elements on the right, we can conclude that the second smallest number in the list must be at the end of the list. However, the remaining 4 numbers can be located anywhere. Thus we have 4! = 24 possible arrangements for the other 4 numbers.

9. The following outline of a decision tree for a list $[x, y, z]$ is given. In each box fill in the appropriate $<$ test. [10 pts]



10. Suppose we start with 2^k max heaps of $2^p - 1$ elements each and we have n additional numbers on the side. We combine these max heaps two at a time by picking one of the numbers as a new root, attaching two max heaps as its left and right children, then running `maxheapify` on that root. We continue combining max heaps this way until there is just one big max heap remaining. When combining max heaps we always choose two max heaps of equal size. The number n is exactly as required to make this work.

Answer the following questions (and simplify your answers as much as possible: DO NOT leave any summation):

(a) First: [3 pts]

What is the value of n as a function of p and k?	$2^k - 1$

(b) Second: [3 pts]

How many elements are in the final heap as a function of p and k?	$2^{k+p} - 1$

(c) Third: [8 pts]

In the worst case scenario, what is the total number of swaps that will be made as a function of p and k?	$(1+p)2^k - p - k - 1$

Put scratch work below. Scratch work is not graded but note that you should know how to explain your answer because you may be expected to do this on an exam.

For c, the process of taking a number and maxheapifying it with 2 children heaps, worst case, is p swaps because each swap has p levels. Add a new node at the root, there are $p+1$ levels, which at worst is p swaps. Do this 2^{k-1} times. After this, we have 2^{k-1} heaps of $2^{p+1} - 1$

$$\sum_{i=0}^{k-1} 2^i (p + k - i - 1)$$

$$(p + k - 1)\sum_{i=0}^{k-1} 2^i - \sum_{i=0}^{k-1} i2^i$$

$$(p + k - 1)(2^k - 1) - (k - 2)2^k - 2$$

$$p2^k - p + k2^k - k - 2^k + 1 - k2^k + 2^{k+1} - 2$$

$$(1 + p)2^k - p - k - 1$$