Brandon Vo

Hw3: Due Saturday, June 12, at 11:59pm

1. You work at a hospital's emergency room. Patients come in and you evaluate the severity of their injury by assigning a **real** number from 1 to 100. Whoever has the highest number (and in case of ties, whoever arrived first) sees the doctors first. When a doctor is available, no time should be wasted, so if possible you want to *instantly* send in the next patient.
   (a) How will you handle the data, to automate this process? How fast can you find the next patient to be seen?

Construct a max-size heap tree. The root will always be the largest node in the heap and will be the next one to see a doctor while the children will be the largest nodes in their subtrees. Whenever a patient is added to the system, the patient is initially added to the very bottom of the heap tree.

After being added, the patient's number will be assessed and checked with the parent node above the patient. If new patient's number is larger, then the patient will swap placed with his parent node. The patient will recursively compare his/her pain number to his/her parent node's number until it reaches a number that is greater than or equal to the new patient's pain number. If it's equal to, then the original patient remains where he/she is at because that person was there first. The new patient will settle into its new position until the doctor takes a patient in and the reheapification process is initiated.

The process of finding the next patient is O(1) time because the doctor will just take the first person in the list, the root of the max heap.

   (b) State how much time it will take you to process new patients, or to reorganize after a patient is sent through to the doctors.

The next patient to be sent to the doctor will be found in O(1) time because the doctor will only check for the root node. After the patient is sent, the list of patients will have to be reheapified where the tree is recursively swapped to replace the root node of the heap tree and every subtree until the heap is reestablished.

In order to complete a reordering of a tree, the tree has to be traversed and recursively swapped from the root to the leaf node at the bottom of the tree. This requires O(log n) time to reheapify the heap tree.

   (c) Explain what to do if a patient gets worse while they're waiting, meaning you decide that their severity is now greater.

Check if the severity is worse based on the new number the patient feels. Assign this new number to the patient and check if the patient needs to be swapped with the person above him/her. If it's worse but not as bad as the person above him/her, then keep that person in the same position.

Otherwise, we can use recursive swapping to move the patient up the priority heap tree. The patient will recursively check if its new value is greater than the parent node before swapping up until the patient either becomes the new root node or the patient's new priority number is not as high as the parent node above him/her. This will mark the patient's new place in the tree. This process takes O(log n) time because this requires traversing up the height of the heap tree.

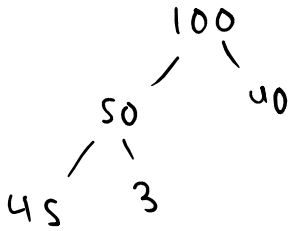
(d) Explain what to do if the evaluation numbers are integers, and how this affects time complexity for the questions above.

We can still compare the pain index numbers regardless if they are real numbers or integers. Because the comparison process doesn't change, the heapification process doesn't change. Time complexity doesn't change because the comparison between integers works the same as the comparison process between real numbers.

Brandon Vo

2. Let [100, 50, 40, 45, 3] be a max-heap that was constructed using the *forward* method (scanning the input left-to-right). Let *x* be the last element that was inserted.

   For each element in the heap, explain why it might be *x*, or why it cannot be.



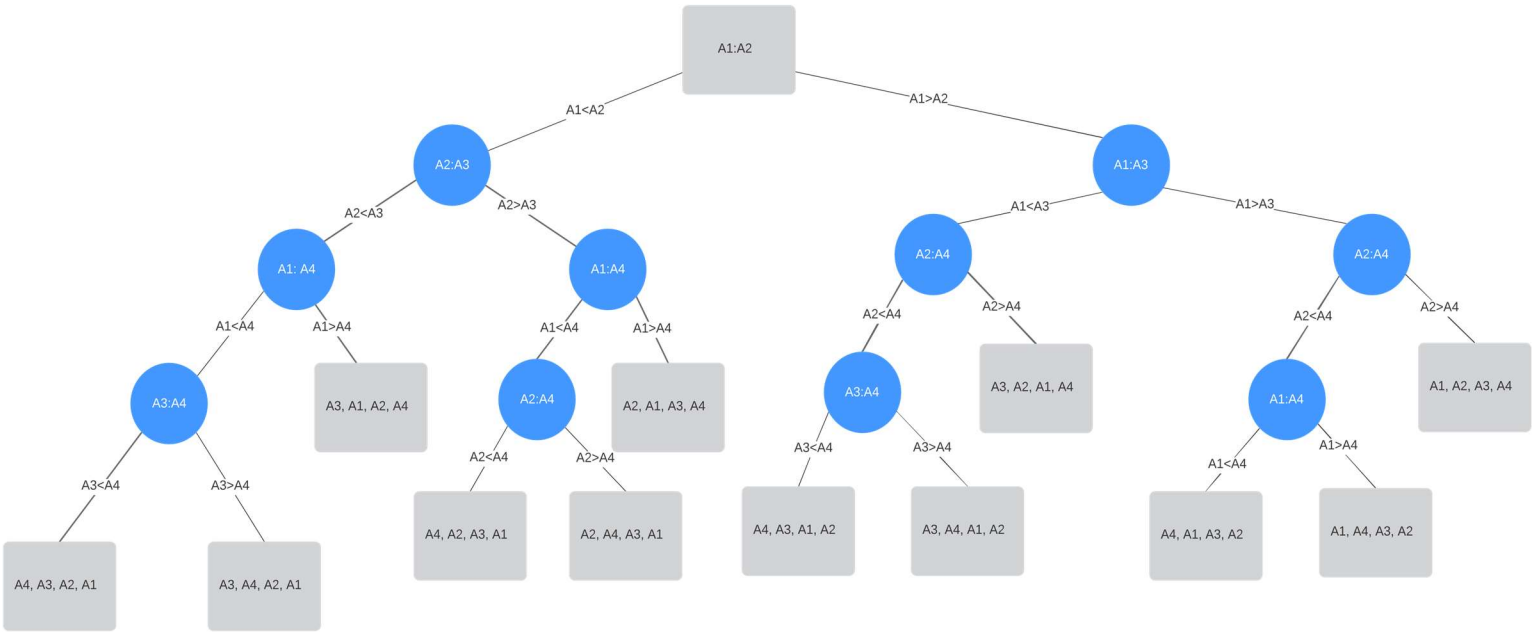
   Above would be the max-heap tree that is constructed using the array provided.

   - Inserting x into the heap, it would be placed at the bottom-leftmost position of the tree, meaning that x has to initially be put as the left child of 40.
   - X would check if it's larger than 40
     - If x is larger than 40, it swaps with 40
     - Else, x does not swap and the heap is established
   - If x has to swap with 40, then it checks with 100
     - If it has to swap with 100, then x would be marked as the largest element in the tree
     - Else, x would remain in 40's original position
       - 40 would be a child node of x
   - X cannot be in the position of 50 or be in the subtree of 50 because 50's sub-heap is already a complete heap. There can be no more children added until 40's subtree is filled out.
     - This excludes the position of [50, 45, 3] from being a possible location of where x could be.

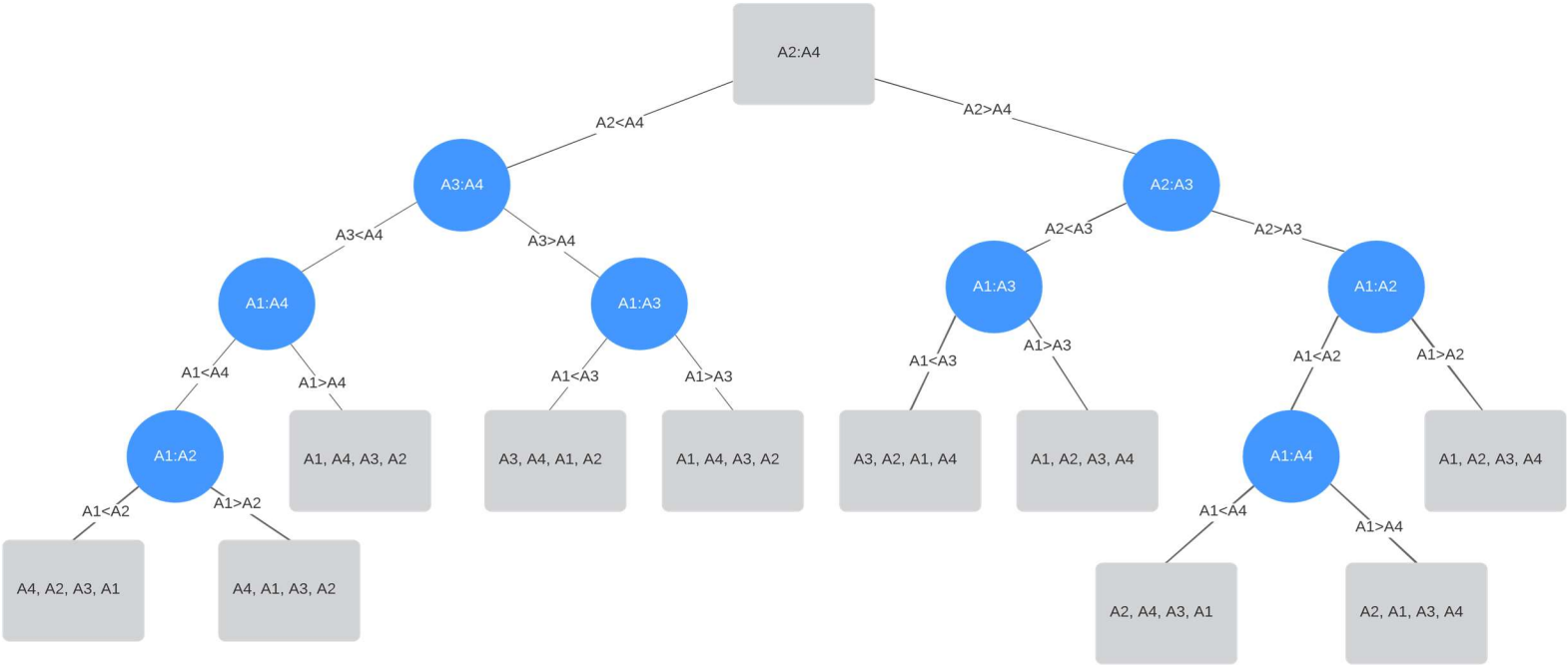
Brandon Vo

3. For an array $A = [a_1,a_2,a_3,a_4]$ of distinct numbers, there are two main ways to build a heap, as described in class. In parts (a) and (b) of this problem you must show what comparisons each method will make, in the form of a binary decision tree. Each leaf should contain output in the form of some permutation of the input subscripts in A (e.g., if you write 3124 it means that after building the heap we have $A = [a_3,a_1,a_2,a_4]$.

(a) Do the above (draw the decision tree) for the forward method.



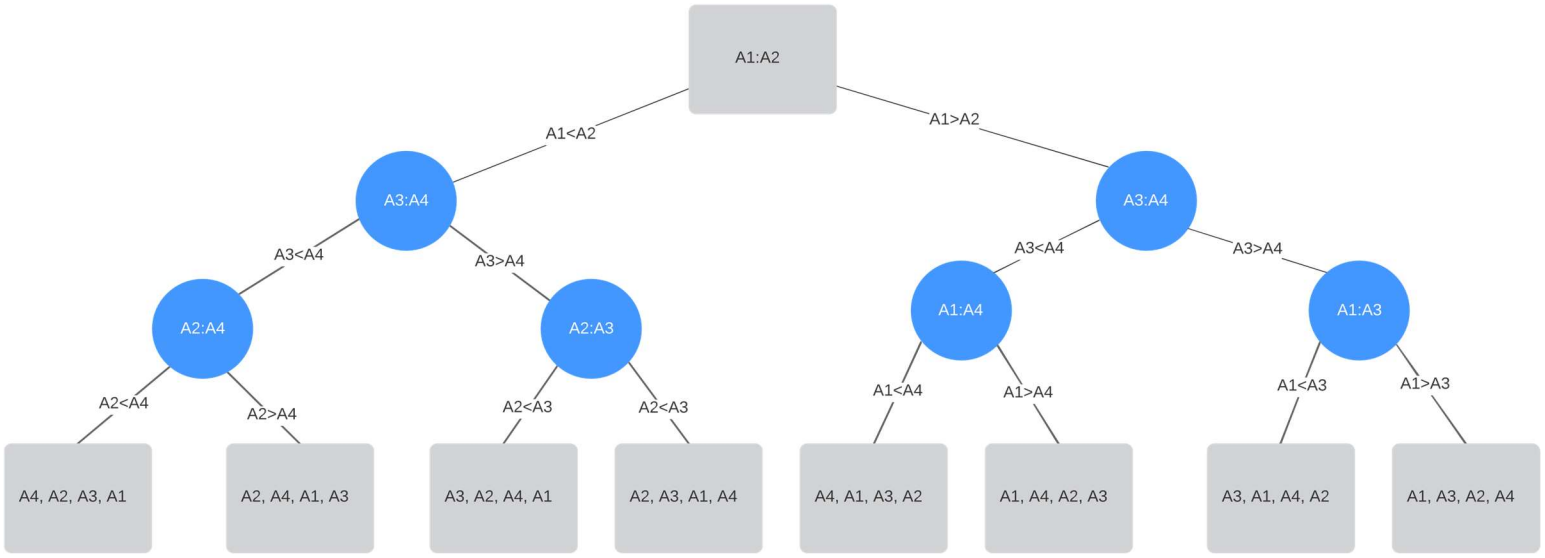
(b) Do the above (draw the decision tree) for the reverse method.



Brandon Vo

(c) Describe your own heap-building algorithm that specifically handles inputs of size 4, and draw the corresponding decision tree that uses fewer decisions in the worst-case compared to the methods in (a) and (b). Your algorithm should be described in English, not pseudocode.

- There are n! input permutations. For the output, there will be atleast $\log_2 4! = \log_2 12$ possible leaves
  - This means there will be a possible height of 3 or 4.
- Similarly to the decision tree, check A1 with A2 and A3 with A4.
  - This splits the size 4 array in half, the best case for optimal decision algorithms
- Of the two halved arrays, take whichever element is the maximum of their sub-arrays
- Of the two maximal elements, check those two elements and see which one is the largest of the two
  - The larger one would be the largest element of the array
  - This element will also be the root node of the heap
  - The other element will be the child of the root node as it is the 2nd-largest element in the array
- Regarding the two lesser elements, the sub-array that has the root node will have its lesser element as the right child of the root node.
  - The last remaining element will be the leaf of the left child of the root, completing the heap.
  - This is because the final element will be compared to the 2nd maximal element of the array, meaning it will never swap up.
- The decision tree below has the best-case scenario of using only 3 decisions needed to yield a result.



Brandon Vo

4. Suppose that we have a set of n distinct numbers that we wish to sort. In class we see that every comparison-sort algorithm must take $\Omega(n\log n)$ time for at least one input permutation, but for all we know there might be an algorithm that is really fast for all other inputs. So now we will show that that's impossible.

   Show that **every** comparison-sort algorithm takes $\Omega(n\log n)$ time, not just in the worst case, but for almost all inputs. Specifically show that the best conceivable algorithm can't even handle $\frac{1}{2^n}$ of all possible inputs in *little-o*$(n\log n)$ time.

   Here $f(n) =$ *little-o*$(n\log n)$ means $f(n) = O(n\log n)$ and $f(n)$ 6= $\Omega(n\log n)$.

   Your answer should not just contain math. You should have an explanation that shows us that you understand exactly what's going on.

- When there are n distinct numbers, that means we have to account for n! possible input permutations
  - For sorting, all possible outputs must be taken into account
- The process to traverse the decision tree to the output would mean traversing the entire height of the decision tree in the worst-case scenario
- Any comparison sorting algorithm, its decision tree must have a minimum of n! nodes to show all possible sorting outputs and all possible arrangements of size n.
- The best comparison-based algorithms do comparisons that split the arrangement into halves. Each further comparison splits the sub-arrangement into further halves until the desired order is found.
  - Because we're splitting our arrays into sub-arrays, we can use only 2 nodes for each parent: 2^n.
  - Considering the $1/2^n$ requirement, decision tree can have only $1/2^n$ of n! leaves based on size n, giving us $n!/2^n$.
  - In a complete tree, the maximum depth will be 2^h leaves based on height
- The time complexity to traverse the height of a tree is in log time.
  - Meaning, it takes $\log_2 n!$ comparisons are needed for n! permutation inputs
    - This means that in order to traverse a tree of n! possible inputs, it would require a depth of atleast $\log_2 \frac{n!}{2^n}$

$$Proof\ by\ contradiction:\ Attempt\ to\ prove\ log\frac{n!}{2^n} \leq 2^h$$

$$\log\frac{n!}{2^n} \leq \log_2 2^h$$

$$\log_2\frac{n!}{2^n} = \log_2 n! - \log_2 2^n \leq h$$

Brandon Vo

$$Stirling\ Formula: \log n! \geq \left(\frac{n}{e}\right)^n$$

$$\log n! \geq \log\left(\frac{n}{e}\right)^n = n\log\left(\frac{n}{e}\right)$$

$$n\log\left(\frac{n}{e}\right) - \log 2^n$$

$$n\log n - n\log e - n, this\ shows\ that\ \log_2 n!\ \ is\ \Omega(\mathrm{n}\log n)$$

$$n\log\left(\frac{n}{e}\right) - \log 2^n = \Omega(\mathrm{n}\log n) - n$$

$$\Omega(\mathrm{n}\log n)\ is\ cn * \log n$$

$$\Omega(\mathrm{n}\log n) - n = cn * \log n - n$$

$$\frac{1}{2}cn * \log n + \left(\frac{1}{2}cn * \log n - n\right) \leq n\log n\ \ when\ n \geq 2\ \ and\ c > 0$$

- This means atleast one leaf will require n log n time to be reached
- Because this is solving for omega, this cannot be o(n log n) time because the sorting algorithm will have a leaf with atleast $\Omega(\mathrm{n}\log n)$ time complexity
- In such a decision tree, there will be two permutations which lead to the same output, but because all values of n are distinct, the two permutations cannot both be in desired order.
    - The double comparisons needed to divide the set of n numbers into halves means that half of the permutation outputs are available as leaves
        - This is because the outputs must be distinct
    - This is why we can use only $\frac{1}{2^n}$ output leaves for the permutation