Answers are formatted in blue and commentary in green. The commentary is intended to provide additional explanations of the answers and/or the grading; you were not expected to provide that.

1. **[10 points]** The complexity function for an algorithm is $\Theta(N \log N)$. Given an input of size $N = 2^{20}$, the running time on a certain computer is about 30 seconds. Haskell Hoo IV offers the opinion that if we doubled the size of the input, the same algorithm, on the same computer, would take about 63 seconds to execute. Is this reasonable? Explain.

Answer:

Let T(N) = operation count for an input of size N. We are given that $T(2^{20})$ operations take about 30 seconds, and that T(N) = N log N. So, we know that

$$T(2^{20}) = 2^{20} \log(2^{20}) = 20 * 2^{20}$$

So, the operation count for the larger input would be about:

$$T(2^{21}) = 2^{21} \log(2^{21}) = 21 * 2^{21} = 20 * 2^{21} + 2^{21} = 2 * 20 * 2^{20} + 2^{21}$$

So, the ratio of the operation counts is:

$$(21 * 2^{21}) / (20 * 2^{20}) = 2 * 21 / 20 = 2.1$$

So, time for the larger input would be about 2.1 * 30 = 63 seconds.

So, Hoo's estimate of 63 seconds seems reasonable.

Full credit required a full analysis that justified more than "about 63 seconds". That is, some answers amounted to saying that an algorithm that's O(N log N) will take a bit more than twice as long if we double the size of the input. That is true:

$$2N \log (2N) = 2N (\log 2 + \log N) = 2N (1 + \log N) = 2N \log N + 2N$$

But that is also imprecise.

Instructions for Q2 - Q9:

[2 points each] An algorithm is, in the average case, $\Theta(N \log N)$. For each part, circle a choice to indicate whether the given property definitely applies to the algorithm, definitely does not apply to the algorithm, or may or may not apply (depending on information not given). No justification is required.

Key observations:

The best case has the average case as an upper bound, but we cannot specify a firm lower bound for the best case.

The worst case has the average case as a lower bound, but we cannot specify a firm upper bound for the worst case.

2. in the best case, $O(N)$ — definitely yes — definitely no — **(maybe yes, maybe no)**

The best case is certainly O(N log N), but O(N) is "smaller"…

3. in the average case, $O(N^2)$ — **(definitely yes)** — definitely no — maybe yes, maybe no

The average case is known to be Θ(N log N), and N^2 is "larger" than that…

4. in the worst case, $\Omega(N)$ — **(definitely yes)** — definitely no — maybe yes, maybe no

The worst case must be Ω(N log N), and N is "smaller" than that…

5. in the worst case, $O(N \log N)$ — definitely yes — definitely no — **(maybe yes, maybe no)**

The worst case is certainly Ω(N log N), and it could even be Θ (N log N), for we know… or not…

6. in the best case, $\Omega(N^2)$ — definitely yes — **(definitely no)** — maybe yes, maybe no

The best case is certainly O(N log N), so it could be Ω (N log N), but no lower bound could be worse than that…

7. in the average case, $\Omega(N \log N)$ — **(definitely yes)** — definitely no — maybe yes, maybe no

Given that the average case is Θ(N log N)…

8. in the worst case, $\Omega(1)$ — **(definitely yes)** — definitely no — maybe yes, maybe no

Well, every algorithm that does anything, must be… best, average, worst case…

9. in the best case, $\Omega(\log N)$ — definitely yes — definitely no — **(maybe yes, maybe no)**

We know the average case is Ω(N log N), the lower bound for the best case could be that or anything smaller…

10. [4 points] Suppose you have a max-heap that contains 10 levels. Assuming that none of the value in the heap are duplicates, where could the smallest value in the heap be located? Consider all relevant possibilities, and justify your answer.

Answer:

First of all, the smallest value obviously must be in a leaf node, since its parent must be larger.

But, where can that leaf be? It depends on whether the bottom level is "full". If so, the smallest value must be in the bottom level; if not, it could be in the level above that (level 8 or level 9 -- we number levels starting at 0).

11. [6 points] True or false: a heap always has the minimum number of levels for the number of nodes it contains. Justify your answer; a single sentence is sufficient (if it's the right sentence).

Answer:

True.

A heap is a complete tree, which means that all levels, except possibly the bottom-most, have a full complement of nodes. Therefore, there would be no empty spaces for a node in the bottom level to be moved into.

Another way of putting it is that a complete tree cannot increase the number of levels until the bottom level of the tree is full.

12. [6 points] In our discussion of the heap data structure, we said that we would map a complete binary tree into an array, by using specific rules relating the index of a parent to the indices of its left and right children.

But, an array is just a linear list, and we could obviously use a doubly-linked list instead of an array to hold the elements in the heap. Are there any vital reasons to not do this? Be precise in justifying your answer.

Answer:

The fact that an array is used gives us O(1) access cost for any array element, if we know its index. If we used a linked list, traversing from a parent to a child would require a linear traversal of the nodes, counting as we go.

That would increase the cost of the algorithms we described, which would be undesirable.

A lot of answers more or less missed the point. We are talking about using a linked list to store the heap, not simply using a linked list to store the same elements. So, the elements would be stored in the same order in both the array and the linked list. The real issue is the cost of accessing an element, given its position (index, if you like).

Full credit only went to answers that explicitly raised the point that we have O(1) random access in an array, but must traverse a section of the linked list to move between parent and child.

13. [8 points] Suppose you are building an application that will involve performing a large number of searches on a huge collection of data records, stored in a file on disk. One type of search will involve a key that is a non-negative integer, an ID number. Each record is guaranteed to have a different value for that key.

You have decided you will handle these searches by building an index in memory, and the index will use some sort of binary search tree for its physical structure. You have chosen a binary search tree, instead of a hash table, because the application will also involve a substantial number of insertions and deletions. We've seen three varieties of BSTs in the course so far.

Research has convinced you that when your application is in use, searches will exhibit high temporal locality, involving a rather small percentage of the data records. What kind of binary search tree should you adopt for your application? Why would you not choose either alternative?

Answer:

The last paragraph suggests that access patterns will be likely to follow a Pareto distribution, rather than a uniform distribution.

That suggests using a splay tree, since that has a chance to keep the most-accessed entries near the root, reducing the average search cost.

Neither the plain BST nor the AVL can do that. A BST holding N elements may have N levels, organized in a fixed manner. An AVL can have no more than 1.5 log N levels, organized in a fixed manner (as far as searches go).

So, the plain BST and the AVL could wind up storing the most-sought elements deep in the tree, and that would drive up the average search cost, since most searches would go to those deep levels.

I hadn't expected answers that considered the heap structure; that's wrong, since a heap is not a binary search tree (although it is a binary tree).