CS 457, Data Structures and Algorithms I
Second Problem Set Solutions

1. **Problem 1** (20 pts) Solve problem 4-3 on page 108 of your textbook. (I will add these soon)

2. **Problem 2** (15 pts) You are given a set S of n integers, as well as one more integer v. Design an algorithm that determines whether or not there exist two distinct elements $x, y \in S$ such that $x+y = v$. Your algorithm should run in time $O(n \log n)$, and it should return (x, y) if such elements exist and (NIL, NIL) otherwise. Prove the worst case running time bound and the correctness of the algorithm.

   Solution:

   There are more than one algorithms that solve this problem within the desired running time bound. One that is simple to write and analyze begins by sorting the elements into a list L in $O(n \log n)$ time using merge-sort or one of the other sorting algorithms with the same worst-case running time bound. Then, it considers every element i from 1 to n and for each one of them it uses binary search for the value $v - L[i]$ in order to search whether the "pair" of $L[i]$ also exists. The correctness of this algorithm is easier to show since it just uses two known sub-routines whose correctness can be taken for granted. The worst-case time bound is also satisfied since the algorithm calls binary search $O(n)$ times and each call needs $O(\log n)$ time. Therefore, the total time of both sorting and searching is $O(n \log n)$.

   What we provide below is a more interesting algorithm that instead completes the searching in linear time instead of $O(n \log n)$. We begin by presenting the algorithm's pseudocode and then provide the analysis.

   Pseudocode:

```
(a)  SUM-EXISTS(S, v){
(b)      L = MAKE-LIST(S)          //Creates a list from a set in O(n) time
(c)      L = MERGE-SORT(L, 1, L.length)        //Merge sort runs in O(n log(n) time
(d)      i = 1
(e)      j = L.length
(f)      while i < j do
(g)          if L[i] + L[j] = v
(h)              return (L[i],L[j])
(i)          else if L[i] + L[j] > v
(j)              j = j - 1
(k)          else
(l)              i = i + 1
(m)      return (NIL, NIL)
(n)  }
```

Proof of Correctness:

Loop Invariant: At the start of each iteration of the loop of lines (f) to (l), no element smaller than $L[i]$ and no element greater than $L[j]$ can be one of the two elements that we are searching for.

Initialization: At the start of the first iteration of this loop $i = 1$ and $j = L.length$. There are clearly no elements smaller than $L[i]$ and no elements greater than $L[j]$ so the loop invariant is vacuously true.

Maintenance: Assume that the loop invariant holds for the k-th iteration. We seek to show that the invariant remains true in the $k + 1$-th iteration. Let l be the value of index i and m be the value of j during the k-th iteration, where $l < m$. During the k-th iteration, the algorithm examined whether $L[l] + L[m] = v$ and, since it continued to enter the $k + 1$-th iteration, this was not the case.

If $L[l]+L[m] > v$, then i remained the same and j decreased by 1 before we enter the $k+1$-th iteration. Since we have assumed our loop invariant holds for $i = l, j = m$, it suffices to show that $L[m]$ could not be one of the elements that we are searching for. Since $L[l] + L[m] > v$, then any number that could be paired with $L[m]$ to give v would have to be smaller than $L[l]$. But, our assumption tells us that no number smaller than $L[l]$ could be one of the two elements that we are searching for. Hence, $L[m]$ could not be one of these elements either.

Similarly, if $L[l]+L[m] < v$, then j remained the same and i increased by 1 before we enter the $k+1$-th iteration. Since we have assumed our loop invariant holds for $i = l, j = m$, it suffices to show that $L[l]$ could not be one of the elements that we are searching for. Since $L[l] + L[m] < v$, then any number that could be paired with $L[l]$ to give v would have to be larger than $L[m]$. But, our assumption tells us that no number larger than $L[m]$ could be one of the two elements that we are searching for. Hence, $L[l]$ could not be one of these elements either.

Termination: There are two possible termination conditions for our algorithm. If it terminates by returning (NIL, NIL), then the last time that the algorith tried to enter the loop it turned out that $i = j$, i.e., both indices point to the same element. In that case, our loop invariant tells us that it must be the case that no number smaller than $L[i]$ and no number greater than $L[i]$ could be one of the elements that we are searching for, and thus no such two elements exist. On the other hand, if our algorithm terminates by returning some pair $(L[i], L[j])$, then lines (g) and (h) of the algorith imply that these two elements satisfy the desired property.

Worst-Case Bound: Since we use the merge-sort algorithm for sorting, its worst-case running time is $O(n \log n)$. Also, while the loop keeps being executed, at each iteration either i is incremented by one or j is decremented by one. Therefore, this loop cannot be executed more than $O(n)$ times, and each iteration takes $O(1)$ time. As a result, the worst-case running time of this algorithm is $O(n \log n) + O(n) = O(n \log n)$.

3. **Problem 3** (15 pts) Prove tight **worst-case** asymptotic upper bounds for the following recurrence equations that satisfy $T(n) = 1$ for $n \leq 2$, and depend on a variable $q \in [0, n/4]$:

Solution:

(a) $T(n) = T(n - 2q - 1) + T(3q/2) + T(q/2) + \Theta(1)$

We begin by guessing that $T(n) \in \Theta(n)$, i.e., that there exist constants c_1 and c_2 such that $c_1 n \leq T(n) \leq c_2 n$ for $n > n_0$. We first prove the upper bound and then the lower bound.

Given our guess, we assume that $T(n') \leq c_2 n'$ for all $n' < n$. This means that

$$T(n) = T(n - 2q - 1) + T(3q/2) + T(q/2) + \Theta(1) \leq c_2(n - 2q - 1) + c_2(3q/2) + c_2(q/2) + \Theta(1).$$

Using the fact that any function in $\Theta(1)$ is at most some constant a, this becomes $T(n) \leq c_2 n - c_2 + a$. If we let $c_2 \geq a$, then this yields $T(n) \leq c_2 n$, which concludes the proof of the upper bound.

For the lower bound, we assume that $T(n') \geq c_1 n'$ for all $n' < n$. This means that

$$T(n) = T(n - 2q - 1) + T(3q/2) + T(q/2) + \Theta(1) \geq c_1(n - 2q - 1) + c_1(3q/2) + c_1(q/2) + \Theta(1).$$

Using the fact that any function in $\Theta(1)$ is at least some constant b, this becomes $T(n) \geq c_1 n - c_1 + b$. If we let $c_1 \leq b$, then this yields $T(n) \geq c_1 n$, which concludes the proof of the lower bound.

(b) $T(n) = T(n - q - 1) + T(n/2 - q) + \Theta(n)$

We start by observing that q appears in both the recursive calls of $T(\cdot)$ with a negative sign. In other words, since $T(n)$ is an increasing function, *no matter what the exact form of* $T(n)$ *is*, the fact that it is an increasing function tells us that the smaller the value of q, the larger the value of $T(n)$. Therefore, unlike the other examples, we can easily verify that the minimum value of q, i.e., $q = 0$ is the one that maximizes $T(n)$. This reduces the recurrence equation to $T(n) = T(n - 1) + T(n/2) + \Theta(n)$.

Now, even after removing the variable q, this recurrence is a bit trickier than the others. In particular, if you guess that $T(n) \in O(n^d)$ for some constant d and then substitute, you will soon run into trouble. The reason for that is the fact that $T(n)$ actually grows faster than any polynomial. To verify this fact, assume that $T(n) \leq cn^d$ for some positive constants c and d. Substitution would then imply that $T(n) \leq c(n - 1)^d + c(n/2)^d$ which, as n goes to infinity, converges to $T(n) \leq c\left(1 + 1/2^d\right) n^d$. But, as $\left(1 + 1/2^d\right) > 1$, this cannot lead to the desired bound that $T(n) \leq cn^d$.

Seeing that $T(n)$ grows faster than any polynomial, we instead guess that it grows exponentially, i.e., that $T(n) \in O(2^n)$. Assuming that $T(n') \leq c2^{n'}$ for all $n' < n$, our goal is to prove that $T(n) \leq c2^n$. Substitution gives

$$T(n) \leq c2^{(n-1)} + c2^{(n/2)} + \Theta(n).$$

Notice that $c2^{n/2} < c2^{n-1} = \frac{c}{2}2^n$. Thus we see that:

$$T(n) \leq \frac{c}{2}2^n + \frac{c}{2}2^n \leq c2^n.$$

This proves that $T(n)$ grows faster than any polynomial and at most exponentially, which would suffice for the purposes of this exercise. I was happy to see though that at least one of you went even further in order to show that $T(n)$ grows at most quasi-polynomially, i.e., that $T(n) = O(n^{O(\log n)})$, which is an even better bound.

(c) $T(n) = T(n-q-1) + T(3q) + \Theta(n)$

Unlike the previous sub-problem, q appears with a negative sign on one recursive call of $T(\cdot)$ and with a positive one in the other. Therefore, we cannot discard this variable before substituting a specific guess regarding $T(n)$. One way of making a good guess for the growth function of $T(n)$ is to replace an extreme value of q (the max or the min value that it can take) or to replace a value in the middle, and then study the recursion tree of the induced variable-free recurrence equation. A different approach would be to start with a large upper bound and then gradually make it smaller and smaller, as long as the substitution proof still works. For this problem, we start by guessing that $T(n) \in \Theta(n^3)$.

We start by assuming that $T(n') \leq cn'^3$ for some positive constant c and all $n' < n$, and our goal is to show that $T(n) \leq cn^3$ as well. Upon substituting, we get

$$T(n) \leq c(n-q-1)^3 + c(3q)^3 + \Theta(n).$$

Since we want to prove an upper bound for $T(n)$ that is true for all possible values of q, we then aim to find the value of q that maximizes the right hand side. In order to do that, we can take a derivative of the right hand side with respect to q in order to study the monotonicity of this function. What we get is

$$dT(n)/dq = -3c(n-q-1)^2 + 81cq^2.$$

Taking the second derivative then yields

$$d^2T(n)/dq^2 = 6c(n-q-1) + 162cq = 6cn + 156cq - 1,$$

which is always positive if $c \geq 1$. This implies that the value of q that maximizes the right hand side of the initial inequality will be one of the two extreme values of q, i.e., either 0 or $n/4$. We can then try both of these values. If we replace with $q = 0$ we get

$$T(n) \;\leq\; c(n-1)^3 + \Theta(n) \;\leq\; c(n^3 - 3n^2 + 3n - 1) + an \;\leq\; cn^3 - (3cn^2 - (3c+a)n + 1),$$

which is at most cn^3 for large enough values of c and n. This proves that $T(n) \in O(n^3)$ when $q = 0$. If we replace $q = n/4$ we get

$$T(n) \;\leq\; c(3n/4-1)^3 + c(3n/4)^3 + \Theta(n) \;\leq\; 2c(3n/4)^3 + an \;=\; 27cn^3/32 + an,$$

which once again is at most cn^3 for large enough values of c and n. This proves that $T(n) \in O(n^3)$ when $q = n/4$ and thus concludes the proof of the upper bound.

It is therefore true that $T(n) \in O(n^3)$, but this bound is not tight. If you try to prove a bound of $O(n^2)$ though, you would realize that it does not hold for $q = n/4$. If you made this observation and provided a lower bound of $\Omega(n^2)$ at this point, which is rather straightforward for $q = n/4$, this would be sufficient for the purpose of this exercise (note that for your lower bound you can choose whatever value of q you want, without explaining that it is the worst-case choice).

If you want to go a bit further though with your upper bound, one thing that you could do is observe that the analysis that we provided above concluding with the fact that the second derivative is positive would still lead to this same conclusion no matter what the exponent of your polynomial was. That is, if you assume that $T(n) \leq cn^d$ for some constant $d \geq 2$, apply the substitution, and take two derivatives, then you would still get a positive second derivative. As a result, *no matter what d is* (!), you can focus on the two cases of $q = 0$ and $q = n/4$. For $q = 0$, our initial recurrence becomes $T(n) = T(n-1) + \Theta(n)$ and it is very easy to see (e.g., by looking at the recursion tree) that it will lead to a bound of $\Theta(n^2)$. For $q = n/4$ on the other hand, you can observe that $T(n) = T(3n/4-1) + T(3n/4) + \Theta(n) \leq 2T(3n/4) + \Theta(n)$, using the fact that $T(n)$ is an increasing function. As a result, we can now use the first case of the master theorem with $\log_b a = \log_{4/3} 2 \approx 2.4$, which will give us a tighter bound of approximately $O(n^{2.4})$.

4. **Problem 4** (10 pts) Consider the following silly randomized variant of binary search. You are given a sorted array A of n integers and the integer v that you are searching for is chosen uniformly at random from A. Then, instead of comparing v with the value in the middle of the array, the randomized binary search variant chooses a random number r from 1 to n and it compares v with $A[r]$. Depending on whether v is larger or smaller, this process is repeated recursively on the left sub-array or the right sub-array, until the location of v is found. Prove a tight bound on the expected running time of this algorithm.

Solution:

For the analysis of the expected running time of this algorithm, we will be using the indicator variable $X_r = \mathbb{I}\{\text{the number between 1 and } n \text{ chosen by the algorithm is } r\}$. Using the choice of r, the algorithm partitions the initial array into two sub-arrays: the length of the left sub-array is $r-1$ and the length of the right sub-array is $n-r$. Since the integer v was chosen uniformly at random from A, it lies in the left sub-array with probability $\frac{r-1}{n}$, in the right sub-array with probability $\frac{n-r}{n}$, and there is also a probability $\frac{1}{n}$ that v is in fact the element in $A[r]$. Since the cost of comparing the value of v with that of $A[r]$ is constant, we can express the running time of the algorithm using the following recurrence equation:

$$T(n) = \sum_{r=1}^{n} X_r \left(\frac{r-1}{n} T(r-1) + \frac{n-r}{n} T(n-r) + \Theta(1) \right)$$

Therefore, the expected running time of the algorithm can be expressed as

$$\begin{aligned}
\mathbb{E}[T(n)] =& \mathbb{E}\left[\sum_{r=1}^{n} X_r \left(\frac{r-1}{n} T(r-1) + \frac{n-r}{n} T(n-r) + \Theta(1) \right) \right] \\
=& \sum_{r=1}^{n} \mathbb{E}\left[X_r \left(\frac{r-1}{n} T(r-1) + \frac{n-r}{n} T(n-r) + \Theta(1) \right) \right] \\
=& \sum_{r=1}^{n} \mathbb{E}\left[X_r\right] \mathbb{E}\left[\frac{r-1}{n} T(r-1) + \frac{n-r}{n} T(n-r) + \Theta(1) \right] \\
=& \sum_{r=1}^{n} \frac{1}{n} \mathbb{E}\left[\frac{r-1}{n} T(r-1) + \frac{n-r}{n} T(n-r) + \Theta(1) \right] \\
=& \sum_{r=1}^{n} \frac{(r-1)\mathbb{E}[T(r-1)] + (n-r)\mathbb{E}[T(n-r)]}{n^2} + \sum_{r=1}^{n} \frac{1}{n} \Theta(1) \\
=& \sum_{r=1}^{n} \frac{2(r-1)\mathbb{E}[T(r-1)]}{n^2} + \Theta(1). \qquad (1)
\end{aligned}$$

This sequence of equations, except the last one, is due to a repeated use of linearity of expectation and of equation (C.24) from your textbook. For a very similar sequence of arguments, see the analysis of RANDOMIZED-SELECT at the top of Page 218. The last equation uses the fact that $\sum_{r=1}^{n}(r-1)\mathbb{E}[T(r-1)]$ is actually equal to $\sum_{r=1}^{n}(n-r)\mathbb{E}[T(n-r)]$.

We now guess that $\mathbb{E}[T(n)] \in O(\log n)$ and we prove it using the substitution method (we present only the upper bound as the lower bound would follow similar structure). In particular, we guess that there exist positive constants c, and n_0 such that $\mathbb{E}[T(n)] \leq c \log n$ for $n \geq n_0$. For the base case, we consider the case $n = 2$ for which $\log 2 = 1$. Since the running time of the algorithm is $\Theta(1)$ for this case, there definitely exists a constant c such that $\mathbb{E}[T(3)] \leq c$. In the rest of the proof, we also show that the inductive step holds.

To prove the upper bound, we assume that $\mathbb{E}[T(n')] \leq c \log n'$ for some constant c and every $n' < n$, which implies that

$$\begin{aligned}
\sum_{r=1}^{n} \frac{2(r-1)\mathbb{E}[T(r-1)]}{n^2} &= \sum_{r=1}^{2} \frac{2(r-1)\mathbb{E}[T(r-1)]}{n^2} + \sum_{r=3}^{n} \frac{2(r-1)\mathbb{E}[T(r-1)]}{n^2} \\
&\leq \frac{2\mathbb{E}[T(2)]}{n^2} + \sum_{r=3}^{n} \frac{2(r-1)c\log(r-1)}{n^2} \\
&\leq \frac{2\mathbb{E}[T(2)]}{n^2} + \sum_{r=3}^{\lceil n/2 \rceil} \frac{2(r-1)c\log(r-1)}{n^2} + \sum_{r=\lceil n/2 \rceil+1}^{n} \frac{2(r-1)c\log(r-1)}{n^2} \\
&\leq \frac{2c}{n^2} + \sum_{r=3}^{\lceil n/2 \rceil} \frac{2(r-1)c\log(n/2-1)}{n^2} + \sum_{r=\lceil n/2 \rceil+1}^{n} \frac{2(r-1)c\log(n-1)}{n^2} \\
&\leq \frac{2c}{n^2} + \frac{n^2 c\log(n/2-1)}{4n^2} + \frac{3n^2 c\log(n-1)}{4n^2} \\
&\leq \frac{2c}{n^2} + \frac{n^2 c\log n - n^2 c\log 2}{4n^2} + \frac{3n^2 c\log n}{4n^2} \\
&\leq c\log n - c/4 + c/(2n^2) \\
&\leq c\log n - c/8.
\end{aligned}$$

The first inequality is just by substitution of the assumption that $\mathbb{E}[T(r-1)] \leq c \log r - 1$. The second one splits the summation in order to obtain an upper bound (see Page 1152 of your textbook). The third inequality uses the fact that $\mathbb{E}[T(2)] \leq \mathbb{E}[T(3)] \leq c$, which we mentioned in our base case, as well as the fact that $\log(r-1)$ is an increasing function, so we can replace r with the largest value that it takes in the corresponding summation. We then use the fact that $\sum_{r=2}^{n/2}(r-1) \leq n^2/8$ and $\sum_{r=n/2}^{n}(r-1) \leq 3n^2/8$. Finally, the last inequality uses the fact that, for $n \geq 3$, we get $c/(2n^2) - c/4 \leq c/16 - c/4 < c/8$.

As a result, using Inequality (1) we conclude that

$$\mathbb{E}[T(n)] \leq c\log n - c/8 + \Theta(1) \leq c\log n - c/8 + a,$$

for some constant a. Choosing a constant $c \geq 8a$ therefore would ensure that $\mathbb{E}[T(n)] \leq c \log n$.

5. **Problem 5** (10 pts) Can the master method be applied to the recurrence $T(n) = 4T(n/2) + n^2 \log n$? Why or why not? Give an asymptotic upper bound for this recurrence.

Solution:

If we try to apply the master theorem, we have $a = 4, b = 2, f(n) = n^2 \log n$. From this, we see that $n^{\log_b a} = n^{\log_2 4} = n^2$. We first note that the first two cases of the master theorem do not apply since $f(n) = n^2 \log n \in \omega(n^2)$. We then also observe that the third case of the master theorem does not apply either, since we get $f(n) \in o(n^{2+\epsilon})$ for every constant $\epsilon > 0$.

Aiming to guess a reasonable solution, we examine the recursion tree of the equation, which would have a height of $O(\log n)$. Furthermore, we see that at any given level of the tree we have four times more nodes than those of the level above it, but each node solves a problem of half the size. Therefore, the total cost at the first level is $n^2 \log n$ and the total cost of the second level is

$$4(n/2)^2 \log n/2 = n^2(\log n - \log 2) = n^2(\log n - 1).$$

This indicates that the total cost at each is approximately $n^2 \log n$. Using this intuition, and the fact that the depth of the tree is $O(\log n)$ we guess that $T(n) = \Theta(n^2(\log(n))^2)$ and assume that $T(n') \leq cn'^2(\log n')^2$ for some $c > 0$ and all $n' < n$. Upon substitution, we get

$$\begin{aligned}
T(n) &\leq 4c(n/2)^2(\log(n/2))^2 + n^2 \log n \\
&= cn^2(\log n - \log 2)^2 + n^2 \log n \\
&= cn^2(\log n - 1)^2 + n^2 \log n \\
&= cn^2((\log n)^2 - 2\log n + 1) + n^2 \log n \\
&= cn^2(\log n)^2 + (1 - 2c)n^2 \log n + cn^2 \\
&\leq cn^2(\log n)^2 + (1 - 2c)n^2 \log n + cn^2 \log n \\
&= cn^2(\log n)^2 + (1 - c)n^2 \log n,
\end{aligned}$$

where the last inequality comes from the fact that $1 \leq \log n$ when $n \geq 2$. But, if we let $c \geq 1$, this implies that $(1 - c)n^2 \log n \leq 0$ and hence $T(n) \leq cn^2(\log n)^2$.

For the base case, it suffices to show that $T(n) \leq cn^2(\log n)^2$ for values of $n \in (1, 2]$. But, since $T(n)$ is constant for such values of n and $n^2(\log n)^2 > 0$, there definitely exist value of c large enough such that these inequalities are satisfied as well.

6. **Problem 6** (10 pts) Use a recursion tree to give an asymptotically tight solution to the recurrence:

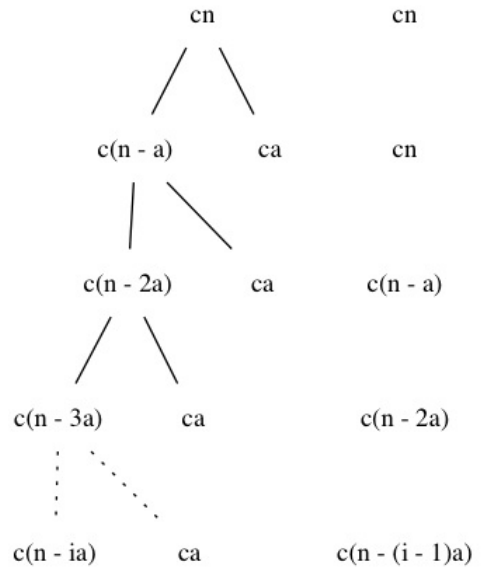
Solution:

(a) $T(n) = T(n-a) + T(a) + cn$, where $a \geq 1$ and $c > 0$ are constants.

At each level of the recursion tree, the new longer (linear) branch has a fewer elements. From this we can say that the total depth of the tree is $\lceil n/a \rceil$, or approximately n/a. Since the smaller branch has cost $O(1)$, we can assume that all the cost is a single node, without loss of generaly. Therefore, the recursion tree would look the the tree in the figure below.

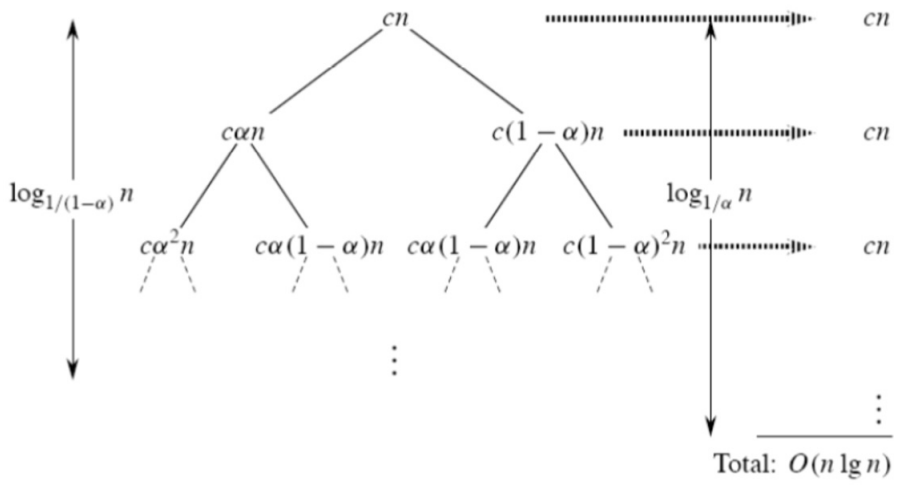
The cost of the long branch at depth d is roughly $c(n-da)$ and the cost of the single node is ca. Summing up over all the levels of the tree,we get

$$\sum_{d=0}^{n/a} c(n-da) + \sum_{d=1}^{n/a} ca \;=\; \sum_{d=0}^{n/a} c(n-da) + cn \;=\; \frac{cn^2}{a} - \frac{n(n/a+1)}{2} + cn \;=\; \Theta(n^2).$$



(b) $T(n) = T(an) + T((1-a)n) + cn$, where $a \in (0,1)$ and $c > 0$ are constants.

We first observe that, at each level of the recursion tree where all branches are still "alive", the cost is cn. The length of the longest branch is either $\log_{1/a} n$ or $\log_{1/(1-a)} n$, depending on whether $a \geq n/2$ or $a < n/2$. Without loss of generality, we assume that $a \geq n/2$ and the other case is symmetric. If $a \geq n/2$, then the length of the longest branch is $\log_{1/a} n$ and the length of the shortest branch is $\log_{1/(1-a)} n$. The important thing is to observe that both the longest and the shortest path are $\Theta(\log n)$ Therefore, even if all the branches remained active as long as the longest one, the total cost would be $O(n \log n)$. But, even if we only measure the cost assuming that all the branches "die" at the same time as the shortest one, then the height remains $\Theta(\log n)$ and thus the total cost is $\Omega(n \log n)$.


cn
cαn
c(1 − α)n
cα²n
cα(1 − α)n
cα(1 − α)n
c(1 − α)²n
log1/(1−α) n
log1/α n
cn
cn
cn
Total: O(n lg n)

7. **Problem 7** (10 pts) For the following recurrences, find the bound implied by the master theorem. Then, try to prove the same bound using the substitution method. If your initial approach fails, show how it fails, and try subtracting off a lower-order term to make it work:

Solution:

(a) $T(n) = 4T(n/3) + n$

Master theorem: To apply the master theorem, we have $a = 4, b = 3$, and $f(n) = n$. Therefore, $n^{\log_b a} = n^{\log_3 4}$, which implies that $f(n) = O(n^{\log_a b - \epsilon})$ for $\epsilon \in (0, \log_3 4 - 1]$. The first case of the master theorem then yields $T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_3 4})$.

Substitution: Knowing that the solution is $T(n) = \Theta(n^{\log_3 4})$, we start by trying to show that there exist constants c and n_0 such that $T(n) \leq cn^{\log_3 4}$ for all $n \geq n_0$. We first assume that our hypothesis holds for all $n < n'$. Making our substitution gives:

$$T(n) \leq 4c(n/3)^{\log_3 4} + n.$$

Unfortunately, this creates a slight difficulty since:

$$4c(n/3)^{\log_3 4} + n = 4c\frac{n^{\log_3 4}}{3^{\log_3 4}} + n = cn^{\log_3 4} + n,$$

which does not imply the desired inequality. We thus modify our initial assumption by subtracting a lower-order term. We now seek to show $T(n) \leq cn^{\log_3 4} - dn$ for some $c, d \geq 0$ and all $n \geq n_0$. We may do this since $n^{\log_3 4}$ is a strictly higher order term than dn and thus $(cn^{\log_3 4} - dn) \in \Theta(cn^{\log_3 4})$. Making our new substitution gives:

$$T(n) \leq 4(c(n/3)^{\log_3 4} - d(n/3)) + n.$$

Rewriting this gives:

$$\begin{aligned} T(n) &\leq 4c(n/3)^{\log_3 4} - \frac{4d}{3}n + n \\ &\leq cn^{\log_3 4} - \left(\frac{4d}{3} - 1\right)n. \end{aligned}$$

For this to imply that $T(n) \leq cn^{\log_3 4} - dn$ it suffices to choose a value of d such that

$$\frac{4d}{3} - 1 \geq d \;\Rightarrow\; 4d \geq 3d + 3 \;\Rightarrow\; d \geq 3.$$

If we let $d = 3$, then for the base case it suffices to show that $T(n) \leq cn^{\log_3 4} - 3n$ for any value of $n \in [1, 3]$ (note that proving the base case for a single value would not make sure that "all the dominoes would fall"). But, since the value of $T(n)$ is constant for all these values of n, it is easy to verify that we can just choose a value of c large enough for the desired inequality to be true.

To complete the proof, we now need to also prove a lower bound. Using an almost identical sequence of steps, we seek to show $T(n) \geq cn^{\log_3 4} - dn$ for two new constants c and d. Making our substitution gives:

$$T(n) \geq 4(c(n/3)^{\log_3 4} - d(n/3) + n$$

which, following the same steps as above, leads to the conclusion that $T(n) \geq cn^{\log_3 4} - dn$ as long as $d \leq 3$. If we let $d = 3$ again, then for the base case it suffices to show that $T(n) \geq cn^{\log_3 4} - 3n$ for any value of $n \in [1, 3]$. But, since the value of $T(n)$ is constant for all these values of n, it is easy to verify that we can just choose a value of c small enough for the desired inequality to be true.

(b) $T(n) = 4T(n/2) + n$

Master theorem: Using the master theorem, we have $a = 4, b = 2, f(n) = n$. $n^{\log_b a} = n^{\log_2 4} = n^2$, which implies that $f(n) = O(n^{\log_a b - \epsilon})$ for $\epsilon \in (0, 1]$. The first case of the master theorem implies that $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$.

Substitution: Knowing that the solution is $T(n) = \Theta(n^2)$, and using a similar line of reasoning as in the first recurrence, we start by subtracting a lower order term and seeking to show $T(n) \leq cn^2 - dn$ for some $c, d \geq 0$ and all $n \geq n_0$. We may do this since cn^2 is a higher order term than dn. Assuming that this holds true for all $n < n'$ and making our substitution gives:

$$T(n) \leq 4(c(n/2)^2 - d(n/2)) + n.$$

Rewriting gives:

$$\begin{aligned} T(n) &\leq 4c(n/2)^2 - 2dn + n \\ &\leq 4c(n/2)^2 - (2d - 1)n. \end{aligned}$$

For this to imply that $T(n) \leq cn^2 - dn$, it suffice to choose a value of d such that

$$2d - 1 \geq d \;\Rightarrow\; d \geq 1.$$

If we let $d = 1$, then for the base case it suffices to show that $T(n) \leq cn^2 - n$ for any value of $n \in [1, 3]$. But, since the value of $T(n)$ is constant for all these values of n, it is easy to verify that we can just choose a value of c large enough for the desired inequality to be true.

To complete the proof, we now need to also prove a lower bound. Using an almost identical sequence of steps, we seek to show that $T(n) \geq cn^2 - dn$ for two new constants. We can make our substitution and get:

$$T(n) \geq 4(c(n/2)^2 - d(n/2)) + n$$

which, following the same steps as above, leads to the conclusion that $T(n) \geq cn^2 - dn$ as long as $d \leq 1$ If we let $d = 1$ again, then for the base case it suffices to show that $T(n) \geq cn^2 - n$ for any value of $n \in [1, 3]$. But, since the value of $T(n)$ is constant for all these values of n, it is easy to verify that we can just choose a value of c small enough for the desired inequality to be true.

8. **Problem 8** (5 pts) Solve the recurrence $T(n) = 3T(\sqrt{n}) + \log n$ by making a change of variable. Your solution should be asymptotically tight. Do not worry about whether values are integral.

Solution:

We first make the change of variable by renaming $m = \log n$. In doing so, we rewrite the equation to be

$$T(2^m) = 3T(2^{m/2}) + m.$$

We can then write $S(m) = T(2^m)$. With this substitution, we find the equation:

$$S(m) = 3T(m/2) + m.$$

To use the master method, we have $a = 3, b = 2$, and $f(m) = m$. Therefore, $\log_b a = \log_2 3 > 1$ and

$$f(m) = O(m^{\log_b a - \epsilon}) = O(m^{\log_2 3 - \epsilon})$$

for any $\epsilon \in (0, \log_2 3 - 1]$, since $f(m) = m$. From the first rule of the master method, we get

$$S(m) = \Theta(m^{\log_2 3}).$$

Having solved the recurrence equation for $S(m)$, we can now find the solution of the original equation using the fact that $m = \log n$:

$$T(n) = T(2^m) = S(m) = \Theta(m^{\log_2 3}) = \Theta(\log n^{\log_2 3}).$$

9. **Problem 9** (5 pts) Using proof by induction, show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap.

Solution:

In an attempt to make this answer helpful, I will first provide an attempt that fails, and then provide the proof that works. Our *failed attempt* tries to prove the desired inequality using induction over the height of the tree starting from the root, which has the maximum height $\lfloor \log n \rfloor$, and then proving that the property remains true for every height from top to bottom, using the inductive step.

FAILED ATTEMPT...

Base Case: Since a heap is a binary tree, we know that the height of the root is $\lfloor \log n \rfloor$, and we also know that there is just a single node at that height, the root. Therefore, it suffices to show that $\lceil n/2^{h+1} \rceil \geq 1$ for $h \leq \lfloor \log n \rfloor$. But then, this implies that

$$\lceil n/2^{h+1} \rceil \;\geq\; \lceil n/2^{\lfloor \log n \rfloor + 1} \rceil \;\geq\; \lceil 1/2 \rceil \;=\; 1,$$

which concludes the proof for the base case.

Inductive Step: We start by assuming that the desired inequality is true for the number of nodes at some height h, and our goal is to prove that it remains true for the number of nodes at height $h-1$. Since a heap is a binary tree, we know that any node at height h has at most two children at height $h-1$. Our assumption above says that the number of nodes at height h is at most $\lceil n/2^{h+1} \rceil$. Therefore, the number of nodes at height $h-1$ can be at most two times that, i.e., at most $2\lceil n/2^{h+1} \rceil$. Now, it would be tempting to argue that $2\lceil n/2^{h+1} \rceil \leq \lceil 2n/2^{h+1} \rceil = \lceil n/2^{(h-1)+1} \rceil$, which would complete the proof, but unfortunately the first inequality is *not true*. To verify this fact note that if $n = 2$ and $h = 1$ then this inequality would imply that $2\lceil 2/2^2 \rceil \leq \lceil 4/2^2 \rceil$, i.e., that $2 \leq 1$.

SUCCESSFUL ATTEMPT...

Our successful attempt proves the induction bottom up using the fact that the number of leaves of a heap are $\lceil n/2 \rceil$. To verify this property of heaps, consider the function PARENT(i) on Page 152 of your textbook that computes that index of i's parent. This implies that the last parent, i.e., the parent of the n-th node has index $\lfloor n/2 \rfloor$, which implies that the remaining $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$ are leaves.

Base Case: We consider the case where $h = 0$, and our goal is to show that for every value of n and any n-element heap, the number of nodes at height 0 is at most $\lceil n/2^{0+1} \rceil = \lceil n/2^1 \rceil = \lceil n/2 \rceil$. But, the nodes at height 0 are the leaves of the heap and, as we showed above, the number of leaves is in fact at most $\lceil n/2 \rceil$, which proves our base case.

Induction Step: We now assume that for any value of n and any n-element heap, the number of nodes at height h is at most $\lceil n/2^{h+1} \rceil$, and we wish to prove that for any value of n, and any n-element heap, the number of nodes at height $h+1$ is at most $\lceil n/2^{h+2} \rceil$. Given any value of n and any n-element heap, we observe that removing all of its $\lceil n/2 \rceil$ leaves leads to a new heap whose number of nodes is $n' = n - \lceil n/2 \rceil = \lfloor n - 2 \rfloor$. In this new heap, the height of each node that was not removed has dropped by 1. As a result, the nodes of height $h+1$ in the original heap are now that nodes of height h in the new heap. Using our assumption, we know that the number of nodes at height h in the n'-element heap that we have created is at most $\lceil n'/2^{h+1} \rceil = \lceil \lfloor n/2 \rfloor / 2^{n+1} \rceil \leq \lceil n/2^{h+2} \rceil$, which completes the proof.