CS 317, Assignment 2

Answers

1. Consider the following algorithms:

```
algorithm DoSomething(A, m, n):
    if n − m = 0 then
        return
    else
        for i = m to n do
            A_{i,j} ← A_{j,i}
            for j = m to n do
                A_{i,j} ← A_{i,j} + i + j
        DoSomething(A, m + 1, n)
```

```
algorithm DoSomethingElse(A, m, n):
    if n − m = 0 then
        return
    else
        for i = m to n do
            A_{i,j} ← A_{j,i} + i
        p ← (n − m)/3
        DoSomethingElse(A, m, m + p)
        DoSomethingElse(A, m+p+1, m+2p)
        DoSomethingElse(A, m + 2p + 1, n)
```

   (a) Write down the recurrence relation for the time complexity of each algorithm. Justify your answer fully.

Answer:

Let $k = n - m$ be the input size.

DoSomething does not do anything for $k = 0$ so we have $T(0) = 0$. For the inductive formula we note that both the outer and the inner for loop iterate k times for an overall time of $\Theta(k^2)$. The input size for the recursive call is one less than the original input size (since m is incremented and n stays the same), so said recursive call takes $T(k-1)$ time. In all $T(k) = T(k-1) + k^2$.

In DoSomethingElse the for loop iterates k times and the input for each recursive call is one third of the original input k so each of those recursive calls take $T(k/3)$ time (and note that we have three such calls). Therefore $T(k) = 3T(k/3) + k$. The base case does not do anything so we have once more $T(0) = 0$.

   (b) Solve the two recurrence relations you just developed *using the characteristic equation technique* and thus give the running time of each algorithm in Θ notation. Note that I am only asking for the complexity and so you do not need to worry about constants.

Answer:

We have $T(k) - T(k-1) = k^2$, so the characteristic equation for the homogeneous part is $r - 1 = 0$ and so $r_1 = 1$. For the non-homogeneous part we have $(r-1)^3 = 0$ ($b = 1$ and

$d = 2$) and so $r_2 = r_3 = r_4 = 1$. We end up with a single solution $r = 1$ with multiplicity 4. Therefore $T(n) = c_1 1^k + c_2 k 1^k + c_3 k^2 1^k + c_4 k^3 1^k = c_1 + c_2 k + c_3 k^2 + c_4 k^3 = \Theta(k^3)$.

For the next recurrence namely, $T(k) = 3T(3/k) + k$, $T(0) = 0$ we change the variable by putting $k = 3^q$ and so $q = \log_3 k$. With $b_q = T(3^q)$ we have $b_q - 3b_{q-1} = 3^q$. For the homogeneous part the characteristic equation is $r - 3 = 0$ and so $r_1 = 3$. For the non-homogeneous part we have $(r-3)^1 = 0$ and so $r_2 = 3$. It turns out that we have a single solution ($r = 3$) with multiplicity 2 and so $b_q = c_1 3^q + c_2 q 3^q = 3^q(c_1 + c_2 q) = \Theta(3^q q)$. That is, $T(k) = \Theta(3^{\log_3 k} \log_3 k) = \Theta(k \log_3 k) = \Theta(k \log k)$.

---

2. Consider a recursive version of insertion sort that goes like this: *To sort a list of size n, sort the last $n-1$ elements recursively, then deal with the first element.*

   (a) Write down the algorithm.

ANSWER:

algorithm ISORT(S, l, h)**:**
- **if** $l < h$ **then**
  - ISORT(S, $l+1$, h)
  - $j \leftarrow l$
  - **while** $j < h \wedge S_j > S_{j+1}$ **do**
    - $S_j \leftrightarrow S_{j+1}$
    - $j \leftarrow j + 1$

To show that ISORT indeed sorts $S_{l\ldots h}$ we proceed by structural induction.

In the base case $l = h$ the sequence contains a single value and so it is trivially sorted. ISORT therefore does not need to do anything, which is indeed the case.

We now assume by induction hypothesis that $S_{l+1\ldots h}$ is sorted after the call ISORT(S, $l+1$, h). Then the invariant at the end of the iteration of the while loop is: S_j is larger than all the values $S_{l\ldots j-1}$, which are all sorted. This is easily proven by induction over j. The base case is trivially true, as there are no values $S_{l\ldots j-1}$. If $S_j \leq S_{j+1}$ then S_{j+1} is larger than S_j which is in turn larger than all the values $S_{l\ldots j-1}$ by induction hypothesis. Since comparison is transitive it follows that S_{j+1} is larger than all the values in $S_{l\ldots j}$; since $S_{l\ldots j}$ is sorted and $S_j \leq S_{j+1}$ then the sequence $S_{l\ldots j+1}$ remains sorted. If on the other hand $S_j > S_{j+1}$ then these two values are swapped, and once this is done we have it the other way around that is, $S_j \leq S_{j+1}$. We end up with the same case as above and so S_{j+1} is once more larger than all the values in $S_{l\ldots j}$ and again the sequence $S_{l\ldots j+1}$ is sorted.

The invariant implies that the sequence $S_{l\ldots h}$ is sorted at the end of the loop as follows: At the end of the loop $S_{1\ldots j}$ is sorted (invariant), $S_j \leq S_{j+1}$ (negated while condition), and $S_{j+1\ldots h}$ is sorted (by the structural induction hypothesis, noting that if a sequence is sorted then so are all its sub-sequences). Therefore the whole sequence $S_{l\ldots h}$ is sorted.

   (b) Write down the recurrence relation for the running time of your algorithm.

ANSWER:

Obviously the input size is $n = h - l$. Clearly $T(0) = 0$ (nothing is done when $n = 0$). Otherwise we perform a recursive call on input size $n - 1$ (the original sequence less the first element) and then the while loop. Clearly in the worst case the loop iterates n time; indeed, this will be the case when $S_j > S_{j+1}$ is always true case in which the loop terminates only when $j = h$. That is, $T(n) = T(n-1) + n$.

(c) Solve the recurrence relation with full details and thus give the running time of your algorithm in Θ notation.

ANSWER:

This is very similar to the first recurrence we solved for Question 1, so we could copy and paste the solution from there with very minor changes. Just for kicks I am going to solve it differently though, using a summing factors guess. We have:

$$\begin{array}{lcl} T(n) & = & T(n-1) + n \\ T(n-1) & = & T(n-2) + (n-1) \\ T(n-2) & = & T(n-3) + (n-2) \\ & \vdots & \\ T(n-i) & = & T(n-i-1) + (n-i) \\ & \vdots & \\ T(0) & = & 0 \\ \hline T(n) & = & n + (n-1) + (n-2) + \cdots + (n-i) + \cdots + 0 \end{array}$$

That is, $T(n) = \sum_{i=0}^{n} i$ and so $T(n) = n(n+1)/2 = \Theta(n^2)$. This being a (however educated) guess we need to verify this result against the original recurrence and I will do so by induction over n: For the base case $T(0) = 0 \times 1/2 = 0$. For the inductive step we note that $T(n) = T(n-1) + n$ and $T(n-1) = (n-1)n/2$ by inductive hypothesis. Therefore $T(n) = (n-1)n/2 + n = (n/2)((n-1) + 2) = n(n+1)/2$, as desired.

---

3. Design an algorithm that received a binary tree and returns true iff that tree is a binary search tree. Establish the correctness and analyze the running time of your algorithm.

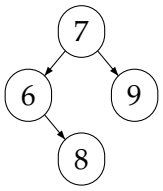
ANSWER:

Note in passing that beside being a question about trees, this is also a typical problem suitable for a divide and conquer approach (like most problems over trees).

It is very tempting to answer this question using the following algorithm (or similar):

```
algorithm DeceptiveCheckBST(T):
    if Empty(T) then return True
    r ← True
    if ¬Empty(Left(T)) then r ← Key(Left(T)) ≤ Key(T)
    if ¬Empty(Right(T)) then r ← r ∧ Key(Right(T)) ≤ Key(T)
    return r ∧ DeceptiveCheckBST(Left(T)) ∧ DeceptiveCheckBST(Right(T))
```

However, this is also incorrect. Indeed, each recursive call checks the root of the respective tree only against the roots of its children, whereas the definition establishes a relationship between the root and *all* the values in the children (not just their roots). Concretely, DeceptiveCheckBST incorrectly returns True on the following (non-binary search) tree:



The only solution to this question I can think of is to traverse the tree in inorder and check that the obtained list is sorted:

```
algorithm CheckBST(T):
    n ← 0
    S ← ⟨⟩
    Inorder(T)
    for i = 1 to n − 1 do
        if S_i > S_{i+1} then return False
    return True
```

```
algorithm Inorder(T):
    if ¬Empty(T) then
        Inorder(Left(T))
        n ← n + 1
        S_n ← Key(T)
        Inorder(Right(T))
```

The for loop in CheckBST is a simple check that $S_{1\ldots n}$ is sorted and so I am not going to bother to prove it correct (too basic). This combined with the fact that an inorder traversal of a binary search tree yields a sorted sequence establishes the correctness of CheckBST. This should be common knowledge (from CS 304), but let me prove that anyway for the record.

I am going to prove by structural induction that Inorder(T) constructs an array $S_{i\ldots i+n}$ that (a) contains all the values in T and (b) is sorted in increasing order iff T is a binary search tree.

For the base case the sequence returned by Inorder is empty, which is clearly sorted (which is correct since an empty tree is always a binary search tree) and contains exactly all the (nonexistent) values in T.

For the inductive step the array constructed by Inorder(T) looks like this: $S_{1\ldots n'}$ is constructed by Inorder(Left(T)), $S_{n'+1}$ = Key(T), and $S_{n'+1\ldots n}$ is constructed by Inorder(Right(T)). Now $S_{1\ldots n'}$ and $S_{n'+1\ldots n}$ contain exactly all the values from Left(T) and Right(T), respectively (by the inductive hypothesis) and in addition to those $S_{1\ldots n}$ also contain Key(T) (in $S_{n'+1}$) so it clearly contains exactly all the values in T.

If T is a binary search tree then so are Left(T) and Right(T). Therefore $S_{1\ldots n'}$ is sorted (inductive hypothesis), is followed by Key(T) which is larger than all the values in $S_{1\ldots n'}$ (by

definition of binary search trees), which is smaller than all the values in $S_{n'+1\ldots n}$ (again, that is how binary search trees work), which is also sorted (by inductive hypothesis). It follows that the whole array $S_{1\ldots n'}$ is sorted, as desired.

Conversely, assume now that T is not a binary search tree. The possible reasons are that either:

(a) $\textsc{Left}(T)$ [or $\textsc{Right}(T)$] is not a binary search tree, case in which $S_{1\ldots n'}$ [or $S_{n'+1\ldots n}$] is not sorted by induction hypothesis and so $S_{1\ldots n}$ is not sorted either.

(b) There is a value in $\textsc{Left}(T)$ larger than $S_{n'+1} = \textsc{Key}(T)$. That value is somewhere in $S_{1\ldots n'}$ (by induction hypothesis), which means that a larger value precedes a smaller value and so $S_{1\ldots n}$ is not sorted.

(c) There is a value in $\textsc{Right}(T)$ smaller than $S_{n'+1} = \textsc{Key}(T)$. That value is somewhere in $S_{n'+2\ldots n}$, which means again that a larger value precedes a smaller value and so $S_{1\ldots n}$ is not sorted.

In all, the array $S_{1\ldots n}$ is not sorted whenever T is not a binary search tree.

The running time for Inorder is tricky if we try to express it as a recurrence. The base case is simple ($T(0) = 0$), but for the general case the best we can do is $T(n) = 1 + T(n') + T(n - 1 - n')$. I cannot make any assumption about the actual value of n' since the tree can have one sub-tree empty, or have the nodes equally divided between the two sub-tress, or (most likely) something in between. This means that I do not know how to solve that recurrence as is.

One way to go about it is to solve the recurrence for the extremes ($n' = n/2$ and $n' = 0$), which will both yield $T(n) = \Theta(n)$ (try it), which makes me confident in guessing that $T(n) = \Theta(n)$ for all possible values of n'. I can then verify my guess by induction over n (again, try it). This is not hard to do, but is very tedious, so instead I am going to use a trick: Each value in T is inserted in S exactly once given the correctness property established above. There are n values in T. Therefore overall $\textsc{Inorder}(T)$ performs exactly n insertions in $\Theta(1)$ time each, for an overall running time in $\Theta(n)$.

After the call to Inorder we have a for loop that clearly iterates $n - 1$ times, so in all the running time of CheckBST is $\Theta(n) + n - 1 = \Theta(n)$. This should be optimal considering that we cannot skip checking any value in T and still be able to tell if it is a binary search tree.

Alternative solution: Most if not all submissions came up with the following different, nicer algorithm:

```
algorithm CheckBSTV2(T):
    CheckBSTRec(T, −∞, +∞)
algorithm CheckBSTRec(T, l, h):
    if Empty(T) then
        return True
    else
        if Key(T) ≤ l ∨ Key(T) ≥ h then return False
        return CheckBSTRec(Left(T), l, Key(T)) ∧ CheckBSTRec(Right(T), Key(T), h)
```

I wish somebody told me where did that come from (nobody actually bothered), as it is highly unlikely that everybody developed the same algorithm independently. In any event,

this algorithm is nicer because it does not use any additional storage, it has the same linear time complexity (one call to CheckBSTRec for each node) and is also correct.

To see the latter we proceed by structural induction. To involve the bounds l and h we must actually prove the following stronger property: CheckBSTRec(T, l, h) returns True iff T is a BST, and all the keys in T are between l and h.

For the base case, an empty tree is obviously a BST and its (nonexisting) keys are obviously between the stated bounds. In this case CheckBSTRec returns True, as it should.

For the inductive step, whenever either of the two recursive calls return False the respective sub-trees are not BST (by inductive hypothesis), and so the current tree is not a BST either. Guess what, the current call returns False as desired. Otherwise, by inductive hypothesis Left(T) is a BST with keys between l and Key(T), and Right(T) is a BST with keys between Key(T) and h. Then T is a BST iff $l \leq \text{Key}(T) \leq h$ and indeed the call returns False iff this is not the case (according to the conditional before the return), again as desired.

---

4. Design a non-recursive algorithm for the CFindset (collapsing find) disjoint set algorithm. Establish the correctness and analyze the running time of your algorithm.

Answer:

I am going to do it in two steps: First we find the root (which we will eventually return). Then we traverse again the path from i to the just found root, setting all the parent links to the root.

```
algorithm CFindset(i):
    root ← i
    while parent_root ≠ root do
        root ← parent_root
    j ← parent_i
    while j ≠ root do
        parent_i ← root
        i ← j
        j ← parent_i
    return root
```

The first loop iterates from the leaf i up along the parent path to the root, so the running time is the same as the height of the tree. The same is true for the second loop as well, though in a bit more complicated a manner (in one iteration i is assigned j and in the next j is assigned the parent of i). Overall the running time is the same as the height of the tree, just like for the recursive version.

The correctness of the first loop is immediate: We just assign *root* to its parent until we can no longer go up (since $parent_{root} = root$). This establishes *root* as the root of the tree and also ensures termination since the path to root is finite and we go up along that path at each iteration.

An obvious invariant at the end of each iteration of the second loop is $parent_i = root \wedge j = parent_i$ (this invariant follows form the assignments inside the loop). The first term establishes that the root of j is collapsed to *root* as intended. The second term ensures that in the next iteration when we do $i \leftarrow j$ we go one step up in the tree. This in turn establishes that (a) we collapse the root for all the nodes j in the path and (b) the loop terminates (since the path to the root is finite).

---

5. The transpose $G^{\mathbb{T}}$ of a directed graph $G = (V, E)$ reverses all its edges that is, $G^{\mathbb{T}} = (V, \{(v, u) \in V \times V : (u, v) \in E\})$. Design efficient algorithms for computing $G^{\mathbb{T}}$ from G for *both* the adjacency-list and adjacency-matrix representations of G. Establish the correctness and analyze the running times of your algorithms.

Answer:

Let as usual $n = |V|$ and $m = |E|$. Also let A and $A^{\mathbb{T}}$ be the matrix representation of G and $G^{\mathbb{T}}$ respectively. It then holds that $A_{i,j} = A^{\mathbb{T}}_{j,i}$ for all $1 \leq i, j \leq n$. Indeed, whenever $A_{i,j} = 1$ the edge $(i, j) \in E$ which implies that $(j, i) \in E^{\mathbb{T}}$, that is, $A^{\mathbb{T}}_{i,j} = 1$. Conversely, $A_{i,j} = 0$ implies that there is no edge (i, j) in G, which means that there is no edge (j, i) in $G^{\mathbb{T}}$ either that is, $A^{\mathbb{T}}_{j,i} = 0$. Therefore $G^{\mathbb{T}}$ can be found using the following straightforward algorithm:

algorithm TransposeMatrix($A; A^{\mathbb{T}}$)**:**
- **for** $i = 1$ **to** n **do**
  - **for** $j = 1$ **to** n **do**
    - $A^{\mathbb{T}}_{i,j} \leftarrow A_{j,i}$

The running time is clearly $\Theta(n^2)$, which I argue to be optimal for the matrix representation since we have to assign a value to each $A^{\mathbb{T}}_{i,j}$ to create $A^{\mathbb{T}}$.

If the graph is represented as an adjacency list I argue that the following straightforward algorithm is also optimal. With L and $L^{\mathbb{T}}$ the adjacency list representation of G and $G^{\mathbb{T}}$, respectively:

algorithm TransposeList($L; L^{\mathbb{T}}$)**:**
- **for** $i = 1$ **to** n **do** $L^{\mathbb{T}}_i \leftarrow$ Nil
- **for** $i = 1$ **to** n **do**
  - **foreach** $j \in L_i$ **do** Insert($i, L^{\mathbb{T}}_j$)

Correctness is pretty straightforward: $j \in L_i$ means that (i, j) is a vertex in G and so (j, i) must be a vertex $G^{\mathbb{T}}$ that is, $i \in L^{\mathbb{T}}_j$. This is quite literally what the foreach loop does.

In terms of running time, we spend $\Theta(n)$ time to create the lists in the first loop. To the second loop now, Insert is a constant time operation (since there is nothing preventing us from inserting at the begining of the list). The foreach loop executes in linear time (it is a simple list traversal). We perform one Insert for every edge in the adjacency list and so the overall running time is $\Theta(m)$. The overall running time is then $\Theta(n + m)$. I argue that this is

once more optimal: The n adjacency lists in $L^{\mathbb{T}}$ must be created no matter what. Afterward The graph $G^{\mathbb{T}}$ has m edges and so we must perform $\Omega(m)$ operations to find them out.

Make sure you review the submission guidelines posted on the course's Web site before submitting. Note in particular that the only acceptable ways to describe an algorithm are pseudo-code (preferred) or actual code. Textual descriptions in particular are not acceptable.