Algorithm Design and Applications

Michael T. Goodrich
Department of Information and Computer Science
University of California, Irvine

Roberto Tamassia
Department of Computer Science
Brown University

Instructor's Solutions Manual

Chapter 1 Algorithm Analysis

Hints and Solutions

Reinforcement

R-1.1 Hint: Recall the method for graphing on a logarithmic scale.

R-1.2 Hint: Consider how it behaves on average.

Solution: The outer loop, for index j, makes n iterations. In $n/2$ of those iterations (for $j < n/2$), the next-inner loop, for index k, makes at least $n/2$ iterations. Finally, for $n/4$ of those iterations (for $k > 3n/4$), the inner-most loop, for index i, makes at least $n/4$ iterations. Thus, the MaxsubSlow algorithm uses at least $n(n/2)(n/4) = n^3/8$ steps, which is $\Omega(n^3)$.

R-1.3 Hint: Determine the place where these two functions cross.

R-1.4 Hint: Determine the place where the two functions cross.

R-1.5 Hint: Use the limit definition.

R-1.6 Hint: Note the similarity of "always" and "worst case."

R-1.7 Hint: When in doubt about two functions $f(n)$ and $g(n)$, consider $\log f(n)$ and $\log g(n)$ or $2^{f(n)}$ and $2^{g(n)}$.

Solution:

$$1/n, 2^{100}, \log\log n, \sqrt{\log n}, \log^2 n, n^{0.01}, \lceil\sqrt{n}\rceil, 3n^{0.5}, 2^{\log n}, 5n, n\log_4 n,$$

$$6n\log n, \lfloor 2n\log^2 n\rfloor, 4n^{3/2}, 4^{\log n}, n^2\log n, n^3, 2^n, 4^n, 2^{2^n}.$$

R-1.8 Hint: The numbers in the first row are quite large.

Solution: The numbers in the first row are quite large. The table below calculates it approximately in powers of 10. People might also choose to use powers of 2. Being close to the answer is enough for the big numbers (within a few factors of 10 from the answers shown).

	1 Second	1 Hour	1 Month	1 Century
$\log n$	$2^{10^6} \approx 10^{300000}$	$2^{3.6\times10^9} \approx 10^{10^9}$	$2^{2.6\times10^{12}} \approx 10^{0.8\times10^{12}}$	$2^{3.1\times10^{15}} \approx 10^{10^{15}}$
$\sqrt{n}$	$\approx 10^{12}$	$\approx 1.3 \times 10^{19}$	$\approx 6.8 \times 10^{24}$	$\approx 9.7 \times 10^{30}$
n	10^6	3.6×10^9	$\approx 2.6 \times 10^{12}$	$\approx 3.12 \times 10^{15}$
$n \log n$	$\approx 10^5$	$\approx 10^9$	$\approx 10^11$	$\approx 10^{14}$
n^2	1000	6×10^4	$\approx 1.6 \times 10^6$	$\approx 5.6 \times 10^7$
n^3	100	≈ 1500	≈ 14000	≈ 1500000
2^n	19	31	41	51
$n!$	9	12	15	17

R-1.9 Hint: We say that an algorithm is linear if its running time is proportional to its ***input*** size.

Solution: The worst case running time of find2D is $O(n^2)$. This is seen by examining the worst case where the element x is the very last item in the $n \times n$ array to be examined. In this case, find2d calls the algorithm arrayFind n times. arrayFind will then have to search all n elements for each call until the final call when x is found. Therefore, n comparisons are done for each arrayFind call. Since arrayFind is called n times, we have $n * n$ operations, or an $O(n^2)$ running time. This is not a linear time algorithm; it is quadratic. If this were a linear time algorithm, the running time would be proportional to its input size.

R-1.10 Hint: Don't forget the base case.

R-1.11 Hint: Note the structure of the loop.

Solution: The Loop1 method runs in $O(n)$ time.

R-1.12 Hint: Note the structure of the looping.

Solution: The Loop2 method runs in $O(n)$ time.

R-1.13 Hint: Note the structure of the looping.

Solution: The Loop3 method runs in $O(n^2)$ time.

R-1.14 Hint: Note the structure of the looping.

Solution: The Loop4 method runs in $O(n^2)$ time.

R-1.15 Hint: Note the structure of the looping.

Solution: The Loop5 method runs in $O(n^4)$ time.

R-1.16 Hint: Recall the definition of the big-oh notation.

R-1.17 Hint: Recall the definition of the big-oh notation.

R-1.18 Hint: Recall the definition of the big-oh notation.

R-1.19 Hint: Recall the definition of the big-oh notation.

R-1.20 Hint: Recall the definition of the big-oh notation.

Solution: By the definition of big-Oh, we need to find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $(n+1)^5 \leq c(n^5)$ for every integer $n \geq n_0$. Since $(n+1)^5 = n^5 + 5n^4 + 10n^3 + 10n^2 + 5n + 1$, $(n+1)^5 \leq c(n^5)$ for $c = 8$ and $n \geq n_0 = 2$.

R-1.21 Hint: Recall the definition of the big-oh notation.

Solution: By the definition of big-Oh, we need to find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $2^{n+1} \leq c(2^n)$ for $n \geq n_0$. One possible solution is choosing $c = 2$ and $n_0 = 1$, since $2^{n+1} = 2 \cdot 2^n$.

R-1.22 Hint: Recall the definition of the little-oh notation.

R-1.23 Hint: Recall the definition of the little-omega notation.

R-1.24 Hint: Recall the definition of the big-omega notation.

Solution: By the definition of big-Omega, we need to find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $n^3 \log n \geq cn^3$ for $n \geq n_0$. Choosing $c = 1$ and $n_0 = 2$, shows $n^3 \log n \geq cn^3$ for $n \geq n_0$, since $\log n >\geq 1$ in this range.

R-1.25 Hint: Recall the definition of the big-oh notation.

R-1.26 Hint: Recall the definition of the big-oh notation.

R-1.27

R-1.28 Hint: Recall the definition of the big-oh notation.

R-1.29 Hint: Note that you can save a comparison here.

R-1.30 Hint: Recall the formula for the Chernoff bound.

R-1.31 Hint: Revisit the reason why 2 cyber-dollars were used in the original proof.

R-1.32 Hint: Use the Chernoff bound.

Creativity

C-1.1 Hint: Change the max-based formulas to if-statements and add variables that "remember" when you update the running maximum.

C-1.2 Hint: Observe the relationship between M and m and note that we can do the operations of the two loops at the same time.

C-1.3 Hint: You can essentially ignore the operations p_i were i is not a multiple of 3.

C-1.4 Hint: Consider an argument based on each bit position.

C-1.5 Hint: Notice the similarity of this equation and the sum of the numbers from 1 to n.

C-1.6 Hint: Recall the way of characterizing a geometric sum.

C-1.7 Hint: Recall how the power function is defined.

C-1.8 Hint: Recall the role of n_0 in the definition of the big-oh notation.

Solution: To say that Al's algorithm is "big-oh" of Bill's algorithm implies that Al's algorithm will run faster than Bill's for all input greater than some nonzero positive integer n_0. In this case, $n_0 = 100$.

C-1.9 Hint: Think of a function that grows and shrinks at the same time without bound.

Solution: One possible solution is $f(n) = n^2 + (1 + sin(n))$.

C-1.10 Hint: Use induction, a visual proof, or bound the sum by an integral.

Solution:

$$\sum_{i=1}^{n} i^2 < \int_0^{n+1} x^2 dx < \frac{(n+1)^3}{3} = O(n^3)$$

C-1.11 Hint: Try to bound this sum term by term with a geometric progression.

C-1.12 Hint: Use the log identity that translates $\log bx$ to a logarithm in base 2.

C-1.13 Hint: First construct a group of candidate minimums and a group of candidate maximums.

C-1.14 Hint: Note how much work is done in each iteration.

C-1.15 Hint: Consider the first induction step.

Solution: The induction assumes that the set of $n-1$ sheep without a and the set of $n-1$ sheep without b have sheep in common. Clearly this is not true with the case of 2 sheep. If a base case of 2 sheep could be shown, then the induction would be valid.

C-1.16 Hint: Look carefully at the definition of big-Oh and rewrite the induction hypothesis in terms of this definition.

C-1.17 Hint: Use a specific constant, c.

C-1.18 Hint: You need to handle the one item that is not matched.

C-1.19 Hint: Consider summing up the elements of A.

Solution: First calculate the sum $\sum_{i=1}^{n-1} = \frac{n(n-1)}{2}$. Then calculate the sum of all values in the array A. The missing element is the difference between these two numbers.

C-1.20 Hint: Try to bound from above each term in this summation.

Solution:

$$\sum_{i=1} n \log_2 i < \sum_{i=1} n \log_2 n = n \log_2 n$$

C-1.21 Hint: Try to bound from below half of the terms in this summation.

Solution: For convenience assume that n is even. Then

$$\sum_{i=1}^{n} \log_2 i/geq \sum_{i=\frac{n}{2}+1}^{n} \log_2 \frac{n}{2} = \frac{n}{2} \log_2 \frac{n}{2},$$

which is $\Omega(n \log n)$.

C-1.22 Hint: Use induction to reduce the problem to that for $n/2$.

C-1.23 Hint: Consider the contribution made by one line.

C-1.24 Hint: Think first about how you can verify that a row i has the most 1's in $O(n)$ time.

Solution: Start at the upper left of the matrix. Walk across the matrix until a 0 is found. Then walk down the matrix until a 1 is found. This is repeated until the last row or column is encountered. The row with the most 1's is the last row which was walked across.

Clearly this is an $O(n)$-time algorithm since at most $2 \cdot n$ comparisons are made.

C-1.25 Hint: Take advantage of the fact that the number of rows plus the number of columns in A is $2n$.

Solution: Using the two properties of the array, the method is described as follows.

- Starting from element $A[n-1, 0]$, we scan A moving only to the right and upwards.
- If the number at $A[i, j]$ is 1, then we add the number of 1s in that column $(i + 1)$ to the current total of 1s
- Otherwise we move up one position until we reach another 1.

The running time is $O(n)$. In the worst case, you will visit at most $2n - 1$ places in the array.

C-1.26 Hint: Apply the multiplication formula directly.

C-1.27 Hint: Recall the multiplication algorithm taught in grade school.

Solution:

C-1.28 Hint: Be sure to handle the checking needed during a remove operation.

C-1.29 Hint: Apply the amortization analysis accounting technique using extra cyber-dollars for both insertions and removals.

C-1.30 Hint: Consider how many cyber-dollars are saved up from one expansion to the next.

Applications

A-1.1 Hint: Note that every division takes $O(n)$ time, but there are a lot of divisions.

Solution: Since r is represented with 100 bits, any candidate p that the eavesdropper might use to try to divide r uses also at most 100 bits. Thus, this very naive algorithm requires 2^{100} divisions, which would take about 2^{80} seconds, or at least 2^{55} years. Even if the eavesdropper uses the fact that a candidate p need not ever be more than 50 bits, the problem is still difficult. For in this case, 2^{50} divisions would take about 2^{30} seconds, or about 34 years.

Since each division takes time $O(n)$ and there are 2^{4n} total divisions, the asymptotic running time is $O(n \cdot 2^{4n})$.

A-1.2 Hint: Recall the methods for doing an experimental analysis.

A-1.3 Hint: Recall the methods for doing an experimental analysis.

A-1.4 Hint: Number each bottle and think about the binary expansion of each bottle's number.

Solution:

A-1.5 Hint: You can determine all the boxes with pearls in $O(\sqrt{n})$ time.

A-1.6 Hint: Try to extend the $O(\sqrt{n})$-touch solution for the previous problem.

A-1.7 Hint: Rewrite the equation as $A[i] = c - A[j]$. Now what are you looking for?

Solution: We can rewrite the equation as $A[j] = c - A[i]$. Create a Boolean array, B, indexed from 0 to $10n$, all of whose elements are initially **false**. For each element, $A[i]$, if $c - A[i] > 0$, set $B[c - A[i]]$ to **true**. Then, for each $A[j]$ in A, check if $B[A[j]]$ is **true**. If any such cell of B is **true**, then the answer is "yes." Otherwise, the answer is "no." The running time of this method is $O(n)$.

A-1.8 Hint: Reverse the array by using index pointers that start at the two ends.

A-1.9 Hint: Consider using Horner's rule, which is mentioned in an earlier exercise in this chapter.

Solution: Initialize your value $x = 0$. Go through S from beginning to end, and, for each digit d, update $x \leftarrow 10x + d$. The running time is $O(n)$.

A-1.10 Hint: Recall how we solved the maximum subarray sum problem.

A-1.11 Hint: Consider using the XOR function.

Solution: Initialize y to 0 and then XOR all the values in A with y. The result will be x.

A-1.12 Hint: Think of functions that you can compute on all the integers in A.

Solution: Compute the sum of all the integers in A and compute the sum of the squares of all the integers in A. Using the identities given in the appendix of this book, we know that, if all the numbers from 1 to n were present, then the first sum would be $n(n+1)/2$ and the second would be $n(n+1)(2n+1)/6$. So if we denote the missing numbers by i and j, and we denote the first sum by a and the second by b, then we know $a = n(n+1)/2 - i - j$ and $b = n(n+1)(2n+1)/6 - i^2 - j^2$. Solving these two equations will give us the values of i and j.

A-1.13 Hint: Try to count in terms of the losers.

Solution: Each time a game is played, one of the teams is sent home. If we start with n times, this means that there are $n - 1$ games played in total. Thus, the total time for doing this simulation is $O(n \log n)$.

A-1.14 Hint: Do a single scan.

A-1.15 Hint: Think about using a "window" that always contains k 1's.

Solution: Scan through A using two pointers, i and j, such that $A[i:j]$ always has k 1's and i is as close to j as possible. Each time you increment j, you need to move it to the next 1 and then move i to get as close to j as possible to maintain $A[i:j]$ having k 1's. Since each operation increments either i or j, we can charge $2n$ cyber-dollars to pay for all operations. Thus, the total running time is $O(n)$.

A-1.16 Hint: Consider applying the principle of induction to this problem.

Solution: Let's apply induction to this problem. Note that if there is exactly 1 cheating husband, then his wife thinks that there are 0 cheating husbands in the town. So, on the day that the mayor makes his announcement, she learns that her husband must be cheating on her, and she poisons him that very night. By induction, if i nights have passed and no husbands have been poisoned, then every wife who thinks that there are exactly i other husbands who are cheaters learns that her husband must be a cheater (for otherwise the wives of those i other husbands would have poisoned their husbands by now). So, on that night, every such wife will poison her husband. Therefore, if there are k cheaters, then they will all be poisoned on the kth night after the mayor's announcement.

A-1.17 Hint: Consider coin flips in pairs.

Solution: Perform coin flips as ordered pairs, that is, repeatedly perform two flips of this coin, where the order of the flips in each pair matters. If both flips are heads (HH) or tails (TT), then discard this trial and repeat the process with another pair of flips. If the ordered pair of flips comes up HT, however, consider this as equivalent to a "0", and if it comes up TH, consider this as equivalent to a "1." Since individual flips are independent, even for a biased coin, an outcome of HT is equal in probability to a TH, no matter how biased heads and tails are individually.

A-1.18 Hint: Adjust the probability of choosing a byte as you go.

Solution: Use a single variable to hold the chosen byte. Choose the first byte with probability 1, the second with probability $1/2$, and so on, so that you choose the ith byte with probability $1/i$. Any time you choose a byte, you use it to replace the byte you had chosen previously. It is easy to show by induction that each byte will have a probability of $1/n$ in the end of being the one chosen.

Chapter 2 Basic Data Structures

Hints and Solutions

Reinforcement

R-2.1 Hint: Think about the order of the indexing for each of the for-loops.

R-2.2 Hint: Think about the order of the indexing for each of the for-loops.

R-2.3 Hint: Review the code for insertAfter(p, e).

R-2.4 Hint: This one is a real puzzler, and it doesn't even use the operators $+$ and $\times$.

Solution: The tree expresses the formula $6/(1-5/7)$.

R-2.5 Solution: It is not possible for the postorder and preorder traversal of a tree with more than one node to visit the nodes in the same order. A preorder traversal will always visit the root node first, while a postorder traversal node will always visit an external node first.

It is possible for a preorder and a postorder traversal to visit the nodes in the reverse order. Consider the case of a tree with only two nodes.

R-2.6

R-2.7 Hint: Try to gain some intuition by drawing a few different binary trees such that all the external nodes have the same depth.

R-2.8

Creativity

C-2.1 Hint: Review how to implement a general list using a linked list.

C-2.2 Hint: Think about the pointers that must be updated in such implementations.

C-2.3 Hint: Consider a combined search from both ends.

C-2.4 Hint: Use one stack for enqueues and the other for dequeues. (You still need to say how and you also need to do the amortized analysis.)

Solution: Name the two stacks as E and D, for we will enqueue into E and dequeue from D. To implement $\mathsf{enqueue}(e)$, simply call $E.\mathsf{push}(e)$. To implement $\mathsf{dequeue}()$, simply call $D.\mathsf{pop}()$, provided that D is not empty. If D is empty, iteratively pop every element from E and push it onto D, until E is empty, and then call $D.\mathsf{pop}()$. For the amortized analysis, charge \$2 to each enqueue, using \$1 to do the push into E. Imagine that we store the extra cyber-dollar with the element just pushed. We use the cyber-dollar associated with an element when we move it from E to D. This is sufficient, since we never move elements back from D to E. Finally, we charge \$1 for each dequeue to pay for the push from D (to remove the element returned). The total charges for n operations is $O(n)$; hence, each operation runs in $O(1)$ amortized time.

C-2.5 Hint: Use one queue as auxiliary storage and keep track of sizes as you are using it.

Solution: To implement a stack using two queues, $Q1$ and $Q2$, we can simply enqueue elements into $Q1$ whenever a push call is made. This takes $O(1)$ time to complete. For pop calls, we can dequeue all elements of $Q1$ and enqueue them into $Q2$ except for the last element which we set aside in a temp variable. We then return the elements to $Q1$ by dequeing from $Q2$ and enqueing into $Q1$. The last element that we set aside earlier is then returned as the result of the pop. Thus, performing a pop takes $O(n)$ time.

C-2.6 Hint: Reduce the problem to that of enumerating all permutations of the numbers $\{1, 2, \ldots, n-1\}$.

C-2.7 Hint: Consider each output number one at a time.

Solution: Let i be the last number in π. Perform enough enqueue and dequeue operations to remove i from Q, and push i onto the stack, S. Then repeat this operation until Q is empty. Finally, pop all the elements from S and enqueue them into Q in this order. The result will be the numbers, $1, 2, \ldots, n$, ordered according to π. The total time to implement this algorithm is $O(n^2)$, since retrieving each number, i, can take $O(n)$ time in the worst case.

C-2.8 Hint: Think about how to extend the circular array implementation of a queue given in the previous chapter.

Solution:

Maintain a capacity variable and a elementCount variable. Also maintain the variables indexFirst and indexLast. Presuming that overflow doesn't occur, insertion at rank 0 involves inserting the element in array position $\mathsf{indexFirst} - 1$

if indexFrist is greater than 0. Otherwise it is inserted at capacity $- 1$. Then indexFirst is updated to reflect the array index the new element was inserted into.

Removal from rank 0 involves incrementing indexFirst mod capacity.

The array index of elemAtRank(x) can be computed as

$$x + \mathsf{indexFrist} \bmod \mathsf{capacity}.$$

C-2.9 Hint: Consider randomly shuffling the deck one card at a time.

C-2.10

C-2.11 Hint: Think about what could be the worst case number of nodes that would have to be traversed to answer each of these queries.

Solution: Pseudocode for these methods is given below.

The worst case running times for these algorithms are all $O(\log n)$ where n is the height of the tree T.

C-2.12 Hint: Derive a formula that relates the depth of a node v to the depths of nodes adjacent to v.

Solution: The idea is to perform a preorder (or postorder) traversal of the tree, where the "visit" action is to report the depth of the node that is currently visited. This can be easily done by using a counter that keeps track of the current depth.

C-2.13 Hint: Try to compute node heights and balance factors at the same time.

Solution: One way to do this is the following: in the ***external*** method, set height and balance to be zero. Then, alter the ***right*** method as follows:

Algorithm right():
  if $v$isInternal(v) **then**
    if $v.leftchild.height > v.rightchild.height$ **then**
      $v.height = v.leftchild.height + 1$;
    else
      $v.height = v.rightchild.height + 1$;
    $v.balance = absval(v.rightchild.height - v.leftchild.height)$;
  printBalanceFactor(v)

C-2.14 Hint: Use the definition to derive a recursive algorithm.

C-2.15 Solution: Examining the Euler tree traversal, we have the following method for finding tourNext(v,α)

- If v is a leaf, then:

```
Algorithm preorderNext(Node v):
    if v isInternal() then
        return v's left child
    else
        Node p = parent of v
        if v is left child of p then
            return right child of p
        else
            while v is not left child of p do
                v = p
                p = p.parent
            return right child of p

Algorithm inorderNext(Node v):
    if v isInternal() then
        return v's right child
    else
        Node p = parent of v
        if v is left child of p then
            return p
        else
            while v is not left child of p do
                v = p
                p = p.parent
            return p

Algorithm postorderNext(Node v):
    if v isInternal() then
        p = parent of v
        if v = right child of p then
            return p
        else
            v = right child of p
            while v is not external do
                v = leftchildofv
            return v
    else
        p = parent of v
        if v is left child of p then
            return right child of p
        else
            return p
```

Algorithm 2.1: Methods for the solution of Exercise C-2.11.

- ◦ if α = **left**, then w is v and β = **below**,
- ◦ if α = **below**, then w is v and β = **right**,
- ◦ if α = **right**, then
  - ∗ if v is a left child, then w is v's parent and β = **below**,
  - ∗ if v is a right child, then w is v's parent and β = **right**.
- If v is internal, then:
  - ◦ if α = **left**, then β = **left** and w is v's left child,
  - ◦ if α = **below**, then β = **left** and w is v's right child,
  - ◦ if α = **right**, then
    - ∗ if v is a left child, then β = **below** and w is v's parent,
    - ∗ if v is a right child, then β = **right** and w is v's parent.

For every node v but the root, we can find whether v is a left or right child, by asking "v=T.leftChild(T.parent(v))". The complexity is always $O(1)$.

C-2.16

C-2.17 **Hint:** Consider a recursive algorithm.

C-2.18 **Hint:** You can tell which visit action to perform at a node by taking note of where you are coming from.

Solution:

C-2.19 **Hint:** Use a stack.

Solution:

C-2.20 **Hint:** Use a queue.

Solution:

C-2.21 **Hint:** Modify an algorithm for computing the depth of each node so that it computes path lengths at the same time.

C-2.22 **Hint:** Use the fact that we can build T from a single root node via a series of n operations that expand an external node into an internal node with two leaf children.

Applications

A-2.1

A-2.2 **Hint:** It helps to know the relative depths of x and y.

Solution: This method runs in $O(d)$ time, where d is the depth of T.

A-2.3 Hint: Redefine the diameter in terms of node heights.

Solution: Let d_{uv} be the diameter of T. First observe that both u and v are tree leaves; if not, we can find a path of higher length, only by considering one of u's or v's children. Moreover, one of u, v has to be a leaf of highest depth. This can be proved by contradiction. Consider a tree and some leaf-to-leaf path P that does not include a leaf of highest depth. Then consider a leaf of greatest depth v; it is always possible to find a new path P' that starts at v and is longer than P. This yields the desired contradiction.

The main idea of the algorithm is to find a leaf v of highest depth and starting from this leaf to keep moving towards the tree's root. At each visited node u (including v, but excluding the tree's root), the height of u's sibling is computed and the current length L_{max} of the longest path in which v belongs is updated accordingly. When we are at T's root, L_{max} contains the diameter of T. The running time of the algorithm is linear.

We can improve on this algorithm, however. Note that the above algorithm can visit all the nodes in the tree twice, once to find the deepest node, and once to find the height subtrees. We can combine these two operations into one algorithm with a little bit of ingenuity and recursion. We observe that given a binary tree T_v rooted at a node v, if we know the diameters and heights of the two subtrees, we know the diameter of T_v. Imagine we took a path from the deepest node of the left subtree to the deepest node of the other subtree, passing through v. This path would have length $= 2 + \mathsf{height}(T_v, T_v.\mathsf{leftChild}(v)) + \mathsf{height}(T_v, T_v.\mathsf{rightChild}(v))$. Further, note that this is a longest path that runs through the root of T_v, since the height of the left and right subtrees is simply the longest path from their respective roots to any of their leafs. Therefore, the longest path in T_v is simply the maximum of longest path in the found in T_v's left and right subtrees, and the longest path running through v. Since this algorithm visits each node in the tree exactly once, and performs a constant amount of operations at each node, it follows that this algorithm runs in $O(n)$

```
Algorithm eulerTour(Tree T, Position v):
    state ← start
    while state ≠ done do
        if state = start then
            if T.isExternal(v) then
                left action
                below action
                right action
                state ←= done
            else
                left action
                state ← on_the_left
                v ← v.leftchild
        if state = on_the_left then
            if T.isExternal(v) then
                left action
                below action
                right action
                state = from_the_left
                v ← v.parent
            else
                left action
                v ← v.leftchild
        if state = from_the_left then
            below action
            state ← on_the_right
            v ← v.right
        if state = on_the_right then
            if T.isExternal(v) then
                state = from_the_right
                left action
                below action
                right action
                v ← v.parent
            else
                left action
                state ← on_the_left
                v ← v.left
        if state = from_the_right then
            right action
            if T.isRoot(v) then
                state ← done
            else
                if v is left child of parent then
                    state ← from_the_left
                else
                    state ← from_the_right
                v ← v.parent
```

Algorithm 2.2: Methods for the solution of Exercise C-2.18.

```
Algorithm inorder(Tree T):
    Stack S ← new Stack()
    Node v ← T.root()
    push v
    while S is not empty do
        while v is internal do
            v ← v.left
            push v
        while S is not empty do
            pop v
            visit v
            if v is internal then
                v ← v.right
                push v
            while v is internal do
                v ← v.left
                push v
```

Algorithm 2.3: Method for the solution of Exercise C-2.19.

```
Algorithm levelOrderTraversal(BinaryTree T):
    Queue Q = new Queue()
    Q.enqueue(T.root())
    while Q is not empty do
        Node v ← Q.dequeue()
        if T.isInternal(v) then
            Q.enqueue(v.leftchild)
            Q.enqueue(v.rightchild)
```

Algorithm 2.4: Method for the solution of Exercise C-2.20.

```
Algorithm LCA(Node x, Node y):
    int x_dpth ← x.depth
    int y_dpth ← y.depth
    while x_dpth > y_dpth do
        v ← v.parent
    while y_dpth > x_dpth do
        y ← y.parent
    while x ≠ y do
        x ← x.parent
        y ← y.parent
    return x
```

Algorithm 2.5: Method for the solution of Exercise A-2.2.