1. Recurrences (18 marks) Provide tight bounds on $T(n)$ for each of the following recurrences. Assume that $T(n)$ is constant for sufficiently small $n$. You may use the Master Theorem, if you can show that it applies. Otherwise, use the alternative techniques we discussed in class to prove the bounds.

(a) (4 marks) $T(n) = 7T(n/3) + n^2$

Answer: This is a divide-and-conquer recurrence with $a = 7$, $b = 3$, $f(n) = n^2$, and $n^{\log_b a} = n^{\log_3 7}$. Since $1 < \log_3 7 < 2$, we have that $n^2 = \Omega(n^{\log_3 7 + \varepsilon})$ for some constant $\varepsilon > 0$, suggesting that Case 3 of the master theorem may apply. To satisfy the regularity condition, we must have that $af(n/b) \leq cf(n)$ for some constant $c < 1$, i.e. that $7(n/3)^2 \leq cn^2 \leftrightarrow 7/9 \leq c$, which is satisfied by, for example, $c = 8/9 < 1$. Thus Case 3 of the master theorem does apply and $T(n) = \Theta(n^2)$.

(b) (4 marks) $T(n) = 4T(n/2) + n^2 \sqrt{n}$

Answer: We have $f(n) = n^2 \sqrt{n} = n^{5/2}$ and $n^{\log_b a} = n^{\log_2 4} = n^2$. Since $n^{5/2} = \Omega(n^{2+1/2})$, we look at the regularity condition in case 3 of the master theorem. We have $af(n/b) = 4(n/2)^2 \sqrt{n/2} = n^{5/2}/\sqrt{2} \leq cn^{5/2}$ or $1/\sqrt{2} \leq c < 1$. Case 3 applies, and we have $T(n) = \Theta(n^2 \sqrt{n})$.

(c) (4 marks) $T(n) = T(n-2) + 2 \log n$

Answer: We build a recurrence table, neglecting ceiling and floor issues:

<table>
<thead>
<tr>
<th>Level</th>
<th>Instance</th>
<th>Work in</th>
<th>Number of</th>
<th>Work in</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size</td>
<td>Stackframe</td>
<td>Stackframes</td>
<td>Level</td>
</tr>
<tr>
<td>0</td>
<td>$n$</td>
<td>2 log $n$</td>
<td>1</td>
<td>2 log $n$</td>
</tr>
<tr>
<td>1</td>
<td>$n-2$</td>
<td>2 log($n-2$)</td>
<td>1</td>
<td>2 log($n-2$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i$</td>
<td>$n-2i$</td>
<td>2 log($n-2i$)</td>
<td>1</td>
<td>2 log($n-2i$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$n/2-1$</td>
<td>2</td>
<td>2 log 2 = 2</td>
<td>1</td>
<td>2 log 2 = 2</td>
</tr>
</tbody>
</table>

Thus the total work $T(n)$ is given by

$$T(n) = 2 \sum_{i=0}^{n/2-1} \log(n-2i) = 2 \sum_{i=1}^{n/2} \log 2i = 2 \sum_{i=1}^{n/2} (\log 2 + \log i) = n + 2 \sum_{i=1}^{n/2} \log i$$

Claim: $\sum_{i=1}^{n/2} \log i \in \Omega(n \log n)$
Proof:
\[
\sum_{i=1}^{n/2} \log i \geq \sum_{i=n/4}^{n/2} \log i \geq \sum_{i=n/4}^{n/4} \log(n/4) = (n/4 + 1) \log n - 2(n/4 + 1) \in \Omega(n \log n).
\]

Claim: \(\sum_{i=1}^{n/2} \log i \in O(n \log n)\)
Proof: \(\sum_{i=1}^{n/2} \log i \leq \sum_{i=1}^{n} \log n = n \log n\)

Thus \(\sum_{i=1}^{n/2} \log i \in \Theta(n \log n)\) and \(T(n) = n + 2\Theta(n \log n) \in \Theta(n \log n)\)

Note that \(T(n) \in O(n \log n)\) can be verified easily by induction. Suppose that \(T(n - 2) \leq c(n - 2) \log(n - 2)\) for some constant \(c > 0, n \geq 3\). Then
\[
T(n) = T(n - 2) + 2 \log n \leq c(n - 2) \log(n - 2) + 2 \log n \leq (cn - 2c + 2) \log n \leq cn \log n \forall c \geq 1.
\]

Proving that \(T(n) \in \Omega(n \log n)\) by induction is more difficult.

(d) (6 marks) \(T(n) = T(n/2) + T(n/4) + T(n/8) + n\)

- Answer: A solution can be derived from a recursion tree:

We can verify that \(T(n) \in O(n)\) by induction, using the substitution method. Our inductive hypothesis is that \(T(n) \leq cn\) for some constant \(c > 0\). We have:
\[
T(n) = T(n/2) + T(n/4) + T(n/8) + n \\
\leq cn/2 + cn/4 + cn/8 + n = 7cn/8 + n = (1 + 7c/8)n \leq cn \text{ if } c \geq 8.
\]

Therefore, \(T(n) = O(n)\).

Showing that \(T(n) \in \Omega(n)\) is easy: \(T(n) = T(n/2) + T(n/4) + T(n/8) + n \geq n\). Since \(T(n) \in O(n)\) and \(T(n) \in \Omega(n)\), we have that \(T(n) = \Theta(n)\).
2. **Building a heap using insertion**, adapted from CLRS 6-1 (20 marks)

The algorithm BUILD-MAX-HEAP described in CLRS builds a heap with the aid of the routine MAX-HEAPIFY. An alternate design (BUILD-MAX-HEAP'), uses MAX-HEAP-INSERT (see below).

(a) (10 marks) Do the procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP’ always create the same heap when run on the same input array? Prove that they do, or provide a counterexample.

- **Answer:** The procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP’ do not always create the same heap when run on the same input array. Consider the following counterexample:

  **Input array A:**
  
  \[
  A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 \end{bmatrix}
  \]

  **BUILD-MAX-HEAP(A):**
  
  ![Diagram of BUILD-MAX-HEAP(A)]

  **BUILD-MAX-HEAP’(A):**
  
  ![Diagram of BUILD-MAX-HEAP’(A)]

(b) (5 marks) What is the best-case running time for the two methods? Justify your answer.

- **Answer:**

  Regardless of input, BUILD-MAX-HEAP calls MAX-HEAPIFY \( \Theta(n) \) times. The minimum running time occurs when MAX-HEAPIFY never recurses, i.e. when the input is already a heap. In this case, each call to MAX-HEAPIFY involves \( \Theta(1) \) work, so that the best-case running time for BUILD-MAX-HEAP is \( \Theta(n) \).

  Similarly, BUILD-MAX-HEAP’ always generates \( \Theta(n) \) calls to HEAP-INCREASE-KEY. The minimum running time occurs when the while condition is never satisfied, i.e. when the input is already a heap. In this case, each call to HEAP-INCREASE-KEY (and hence MAX-HEAP-INSERT) involves \( \Theta(1) \) work, so that the best-case running time for BUILD-MAX-HEAP’ is also \( \Theta(n) \).

(c) (5 marks) What is the worst-case running time for the two methods? Justify your answer.

- **Answer:**

  From CLRS, Section 6.3 and lecture slides, the worst case running time of BUILD-MAX-HEAP is \( O(n) \). Since we have shown that the best case running time is \( \Theta(n) \), it follows that the worst case running time is \( \Theta(n) \). Hence it can be said more generally that the running time of BUILD-MAX-HEAP is \( \Theta(n) \).
Now consider BUILD-MAX-HEAP'. Each call to HEAP-INCREASE-KEY involves $O(\log n)$ time, since in the worst-case a key must be promoted from the bottom to the top of the tree. An upper bound of $O(n \log n)$ time follows immediately from there being $n - 1$ calls to MAX-HEAP-INSERT. To determine a lower bound, consider the case in which the input array is given in strictly increasing order. Each call to MAX-HEAP-INSERT causes HEAP-INCREASE-KEY to go all the way up to the root. Since the depth of node $i$ is $\lfloor \log i \rfloor$, the total time is:

\[
\sum_{i=1}^{n} \Theta(\lfloor \log i \rfloor) \geq \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \log \lceil n/2 \rceil \rfloor) \\
\geq \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \log (n/2) \rfloor) \\
= \sum_{i=\lceil n/2 \rceil}^{n} \Theta(\lfloor \log n - 1 \rfloor) \\
\geq \frac{n}{2} \Theta(\log n) \\
= \Omega(n \log n).
\]

In the worst case, therefore, BUILD-MAX-HEAP' requires $\Theta(n \log n)$ time to build an n-element heap.

**BUILD-MAX-HEAP(A)**

heap-size[A] = length[A]
for $i = \lfloor \text{length}[A]/2 \rfloor$ downto 1
    MAX-HEAPIFY(A,i)

**BUILD-MAX-HEAP'(A)**

heap-size[A] = 1
for $i = 2$ to length[A]
    MAX-HEAP-INSERT(A, A[i])

**MAX-HEAPIFY(A, i)**

$l$=LEFT($i$)
$r$=RIGHT($i$)
if $l \leq \text{heap-size}[A]$ and $A[l] > A[i]$
    $largest = l$
else
    $largest = i$
if $r \leq \text{heap-size}[A]$ and $A[r] > A[largest]$
    $largest = r$
if $largest \neq i$
exchange $A[i] \leftrightarrow A[largest]$

MAX-HEAPIFY($A, largest$)

MAX-HEAP-INSERT($A, key$)

heap-size[$A$] = heap-size[$A$] + 1

$A[heap-size[A]] = -\infty$

HEAP-INCREASE-KEY($A, heap-size[A], key$)

HEAP-INCREASE-KEY($A, i, key$)

if $key < A[i]$

error “new key is smaller than current key”

$A[i] = key$

while $i > 1$ and $A[PARENT(i)] < A[i]$

exchange $A[i] \leftrightarrow A[Parent(i)]$

$i = PARENT(i)$
3. Binary search trees (15 marks)
You are to design an efficient recursive algorithm that will return values in a binary search tree that bracket a specified key. Specifically, given a binary search tree \( A \) containing \( n \) elements, a key \( k \) and a constant integer \( b \), your algorithm will return two sets \( S_- \) and \( S_+ \). \( S_- \) is to contain the largest \( b \) elements of \( A \) that are smaller than or equal to \( k \). If there are fewer than \( b \) elements smaller than or equal to \( k \), \( S_- \) should contain all of them. \( S_+ \) is defined similarly, but will contain the smallest \( b \) elements that are strictly larger than \( k \).

(a) (10 marks) Provide pseudocode for your algorithm. Clearly specify the precondition and post-condition in your pseudocode. Clearly indicate the base case and how it is handled. How many ‘friends’ are you using? Are they being given a smaller input?

Answer:

algorithm \( \text{Bracket}(T, \text{key}, m) \)

\(<\text{pre-cond}>: T \text{ is a BST, key is a real number, } m \text{ is an integer}

\(<\text{post-cond}>: \text{The } m \text{ largest keys in } T \text{ less than or equal to } \text{key} \text{ are returned in } S_- . \text{ (If } T \text{ contains fewer than } m \text{ keys less than or equal to } \text{key}, \text{ only these are returned.) Similarly, the } m \text{ smallest keys in } T \text{ strictly greater than } \text{key} \text{ are returned in } S_+. \text{ (If } T \text{ contains fewer than } m \text{ keys strictly greater than } \text{key}, \text{ only these are returned.)}\n
begin
1 if \( T = \emptyset \) then return \((\emptyset, \emptyset)\)
2 else if \( T.key \leq \text{key} \)
3 \( (S_-, S_+) = \text{Bracket}(T.right, \text{key}, m) \) %Friend 1
4 if \( |S_-| < m \)
5 \( S_- = S_- \cup T.key \)
6 \( (S_-^*, S_+^*) = \text{Bracket}(T.left, \text{key}, m - |S_-|) \) %Friend 2
7 \( S_- = S_- \cup S_-^* \)
8 end
9 else if \( T.key > \text{key} \)
10 \( (S_-, S_+) = \text{Bracket}(T.left, \text{key}, m) \) %Friend 1
11 if \( |S_+| < m \)
12 \( S_+ = S_+ \cup T.key \)
13 \( (S_-^*, S_+^*) = \text{Bracket}(T.right, \text{key}, m - |S_+|) \) %Friend 2
14 \( S_+ = S_+ \cup S_+^* \)
15 end
16 end
17 return \((S_-, S_+)\)
end algorithm

The base case occurs when the tree \( T \) is \( \emptyset \), i.e., when we hit the bottom of the tree, and empty sets are returned for \( S_- \) and \( S_+ \). Between 1-2 friends are used. Each is given either
the left or right subtree of size \((n - 1)/2\).

**Proof of correctness:**

**Base Case:** If the tree is empty, then \(S_-\) and \(S_+\) are empty, as required.

**General Case:** Suppose that \(T.key \leq key\). Then the right subtree will contain all candidate values for \(S_+\) and also the biggest values less than or equal to \(key\), i.e., the ‘top’ candidates for \(S_-\), and these will be returned to \((S_-, S_+)\) by the first recursive call in Line 3 (‘Friend 1’). Then, if \(|S_-| < m\), we wish to add the largest values from the remainder of the tree. By the structure of BSTs, the largest of these will be \(T.key\), and this will be added in Line 5. Line 6 will then return in \(S_-^*\) the largest \(m - |S_-|\) values in the left subtree that are less than or equal to \(key\), or all values in the left subtree that are less than or equal to \(key\), if there are fewer than \(m - |S_-|\). These are added to \(S_-\) in Line 7. Note that \(S_-^*\) serves as a dummy variable here.

The proof of correctness is analogous for the case \(T.key > key\).

(b) (5 marks) Specify and solve the recurrence relation for the running time \(T(n)\) of your algorithm.

- **Answer:** We are interested in asymptotic performance, and \(m\) is a constant. Thus we can assume for the purposes of asymptotic analysis that \(m << n\).

  Note that the algorithm first uses Friend 1 to descend to the location at the bottom of the BST where \(key\) would be inserted if this was our task. Then as the recursion unwinds it begins to climb back up, adding keys to \(S_-\) and \(S_+\) as it goes.

  On its return climb, it will use Friend 2 to make additional descents left or right, trying to fill \(S_-\) and \(S_+\).

  Since in the worst case two friends are called, and constant work is done in the stackframe, one might be tempted to use the recurrence relation \(T(n) = 2T(n/2) + c\), which is dominated by base cases, and which, by the Master Theorem, yields \(T(n) \in \Theta(n)\).

  The error here is that Friend 2 is only called when \(|S_-| < m\) or \(|S_+| < m\), and this will only occur a constant number of times if the tree is balanced.

  To see this, note that when Friend 2 is called on Line 6, it is passed a subtree that we know contains only values smaller than or equal to \(key\). Thus if Friend 2 is called from height \(\lceil \log(m + 1) \rceil\) it is guaranteed to fill \(S_-\).

  Similarly, when Friend 2 is called on Line 13, it is passed a subtree that we know contains only values greater than \(key\). Thus if Friend 2 is called from height \(\lceil \log(m + 1) \rceil\) it is guaranteed to fill \(S_+\).

  Since Friend 2 is called only as the recursion unwinds up the depth-first path, this condition will be met after at most \(m\) calls to Friend 2.

  Thus for a balanced tree the recurrence relation is \(T(n) = T(n/2) + c\), and the running time is \(\Theta(\log n)\).

  For a worst-case unbalanced tree, it will take \(\Theta(n)\) time just to reach the bottom of the tree, so the algorithm will be \(\Theta(n)\).
4. Signal Detection (42 marks)

A communications channel carries a 1 GHz digital (binary) time signal $A(t)$, i.e., $A(t) \in \{0, 1\}, t \in [1, 2, ..., n]$ nanoseconds. An edge is defined to be a transition of the signal from one state to another. The edge may be of positive polarity, i.e., a transition from 0 to 1, or of negative polarity, i.e., a transition from 1 to 0. The delay of the edge is defined as the midpoint of the transition. For example, if $A[7] = 0$ and $A[8] = 1$ then the delay of the edge is $t = 7.5$ nsec.

Given a time signal $A[1...n]$, you are asked to write an efficient algorithm for time-stamping an edge in $A$, i.e., reporting its delay. If there is more than one edge in $A$, the delay of any one may be reported. If no edges are present in $A$, this fact should be reported.

(a) (5 marks) Show that the problem is $\Omega(n)$, i.e., that all correct algorithms are $\Omega(n)$. Hint: Imagine this as a game in which some adversary gives you a sub-linear algorithm and you must find an input array $A$ for which this algorithm yields the wrong answer.

* Answer: The input instance we consider will either be all 0s or have exactly one 1. Suppose that a correct sublinear algorithm exists (and is given to you by your “adversary”). We start by giving the algorithm an input consisting of all 0s, and observing the array locations the algorithm examines before halting. If the algorithm examines all $n$ locations, the algorithm is not sublinear as claimed and hence the problem is $\Omega(n)$. Alternatively, suppose the algorithm does not examine location $i$. If the algorithm reports the wrong answer, the algorithm is not correct as claimed and the problem is $\Omega(n)$. If the algorithm reports the right answer (no edge present), it will report the wrong answer on the same input but with bit $i$ set.

(b) (5 marks) Now suppose that front-end processing guarantees that the first and last value of the signal are different, i.e., $A[1] \neq A[n]$. Design a recursive sublinear time algorithm `findedge` which reports the delay of an edge. Clearly state the pre- and post-conditions of your algorithm and verify that they are met on each recursive call. Also verify that the size of the input declines monotonically as a function of the depth of recursion. Use pseudocode to describe your algorithm.


algorithm y=findedge(A,p,q)
< pre-cond >: $A[p...q]$ is a bit array with $A[p] \neq A[q]$
< post-cond >: $y$ is the location of an edge in $A$

begin
if $q - p < 2$
\[ y = \frac{(p + q)}{2} \]

else
\[ r = \lfloor \frac{(p + q)}{2} \rfloor \]

if \( A(p) \neq A(r) \)
\[ y = \text{findedge}(A, p, r) \]
else
\[ y = \text{findedge}(A, r, q) \]
end algorithm

(c) (2 marks) Write a recurrence equation for your algorithm.

- Answer: \( T(n) = T(n/2) + \Theta(1) \)

(d) Determine tight asymptotic bounds for your algorithm using

i. (5 marks) The method of substitution

- Answer: We rewrite the recurrence relation as \( T(n) = T(n/2) + d \), where \( d \) is some positive constant. We guess (e.g. by using a recursion tree) that the solution is \( T(n) \in \Theta(\log n) \). To show that \( T(n) \in O(\log n) \), our inductive hypothesis is that \( T(n) \leq c \log n \) for some constant \( c > 0 \). Then
\[
T(n) = T(n/2) + d \leq c \log n/2 + d = c \log n - c + d \leq c \log n \text{ if } c \geq d.
\]
Thus \( T(n) \in O(\log n) \).

To show that \( T(n) \in \Omega(\log n) \), our inductive hypothesis is \( T(n) \geq c \log n \) for some constant \( c > 0 \). Then
\[
T(n) = T(n/2) + d \geq c \log n/2 + d = c \log n - c + d \geq c \log n \text{ if } c \leq d.
\]
Thus \( T(n) \in \Omega(\log n) \).

Since \( T(n) \in O(\log n) \) and \( T(n) \in \Omega(\log n) \), \( T(n) \in \Theta(\log n) \).

ii. (5 marks) The recursion tree method

<table>
<thead>
<tr>
<th>Level</th>
<th>Instance Size</th>
<th>Work in Stackframe</th>
<th>Number of Stackframes</th>
<th>Work in Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( n )</td>
<td>( \Theta(1) )</td>
<td>1</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>1</td>
<td>( n/2 )</td>
<td>( \Theta(1) )</td>
<td>1</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>( \log n )</td>
<td>1</td>
<td>( \Theta(1) )</td>
<td>1</td>
<td>( \Theta(1) )</td>
</tr>
</tbody>
</table>

Thus the total work \( T(n) \) is given by
\[
T(n) = \sum_{i=0}^{\log n} \Theta(1) \in \Theta(\log n).
\]
iii. (5 marks) The master method

- Answer: Since \( a = 1 \), \( b = 2 \) and \( f(n) \in \Theta(1) \), We have \( n^{\log_b a} = n^{\log_2 1} = 1 \in \Theta(f(n)) = \Theta(1) \). Thus \( T(n) = \Theta(n^{\log_b a \log n}) = \Theta(\log n) \).

(e) (2 marks) If \( A[1] = 0 \) and \( A[n] = 1 \), there is at least one positive edge. Does your algorithm ever report the delay of a negative edge for this input?

- Answer: Since our binary search algorithm maintains \( A[p] = 0 \) and \( A[q] = 1 \), only a positive edge will be detected.

(f) (5 marks) Suppose that front-end processing guarantees that \( A[1] = 0 \) and \( A[n] = 1 \) and you must report the delay of a negative polarity edge, if it exists. What is the time complexity of this problem? Please justify your answer.

- Answer: The problem is \( \Omega(n) \), and the proof is almost identical to that for Part (a). For this problem, locations 1 and \( n \) need not be examined, since these are fixed. Now suppose that a correct sublinear algorithm exists. We start by giving the algorithm the input \( A[i] = 0, i \in [1, ..., n - 1], A[n] = 1 \). Note that for this input, location \( n - 1 \) also does not need to be examined, since setting that bit will not generate a negative polarity edge. We observe the array locations the algorithm examines before halting. If the algorithm examines all locations from 1...\( n - 2 \), the algorithm is not sublinear as claimed and hence the problem is \( \Omega(n) \). Alternatively, suppose the algorithm does not examine location \( i \in [1, ..., n - 2] \). If the algorithm reports the wrong answer, the algorithm is not correct as claimed and the problem is \( \Omega(n) \). If the algorithm reports the right answer (no negative polarity edge present), it will report the wrong answer on the same input but with bit \( i \) set.

(g) (5 marks) You are now asked to write an efficient algorithm for estimating the duration of any pulse in the binary time signal. For example, the signal \( ...00000111000... \) contains a positive pulse of duration 3 nsec. The signal may contain zero or more pulses. Show that, in general, no sublinear algorithm exists for this problem. Does the problem become easier if the boundary conditions \( A[1] \) and \( A[n] \) are fixed?

- Answer: This problem is also \( \Omega(n) \). The input instance we consider will either be all 0s or have exactly one 1. Suppose that a correct sublinear algorithm exists. We start by giving the algorithm an input consisting of all 0s, and observing the array locations the algorithm examines before halting. For this input, locations 1 and \( n \) need not be examined, since a solitary 1 in either location will generate an edge, not a pulse. We observe the array locations the algorithm examines before halting. If the algorithm examines all locations from 2...\( n - 1 \), the algorithm is not sublinear as claimed and hence the problem is \( \Omega(n) \). Alternatively, suppose the algorithm does not examine location \( i \in [2, ..., n - 1] \). If the algorithm reports the wrong answer, the algorithm is not correct as claimed and the problem is \( \Omega(n) \). If the algorithm reports the right answer (no pulse present), it will report the wrong answer on the same input but with bit \( i \) set.

Note that the complexity of the problem is independent of the boundary conditions.
(h) (5 marks) Suppose that front-end processing guarantees that \( A[1] = A[n] \), that the signal contains at most one pulse, and that this pulse, if it exists, is at least \( n/a \) nsec in duration, where \( a > 1 \) is a constant. Design an algorithm that uses your \textit{findedge} routine to solve the problem in sublinear time. What is the asymptotic running time of your algorithm?

- Answer: We sample the signal at regular intervals of \( n/a \), just finely enough to ensure that one sample will land on the pulse, if it exists. Note that the maximum number of samples \( m \) is a constant: \( m \approx \frac{n}{n/a} = a \).

\begin{algorithm}
\textbf{algorithm} y=\text{findpulse} (A, n, a)\\
\begin{small}
\text{\underline{pre-cond}:} \ A[1..n] \text{ is an array of bits with } A[1] = A[n]. \ a \text{ is a positive real constant. } A \text{ contains at most one pulse, which must be at least } n/a \text{ in duration. } \\
\text{\underline{post-cond}:} \ y = 0 \text{ if the signal contains no pulse, otherwise } y \text{ is the duration of a pulse in } A \\
\begin{align*}
\text{begin} & \\
\quad r & = 2 \\
\quad dr & = \lceil n/a \rceil \\
\quad \text{while } (r < n) & \& (A(r) == A(1)) \\
\quad & \quad r = r + dr \\
\quad \text{end} & \\
\quad \text{if } r \geq n & \quad y = 0 \\
\quad \text{else } & \\
\quad & \quad y = \text{findedge}(A, r, n) - \text{findedge}(A, 1, r) \\
\text{end algorithm} \\
\end{align*}
\end{small}
\end{algorithm}

The maximum number of samples \( m \) is: \( m = \frac{n-3}{n/a} + 1 \approx a + 1 \in \Theta(1) \). Thus \( T(n) = \Theta(1) + \Theta(\log(n - r + 1)) + \Theta(\log r) \in \Theta(\log n) \).