Problem Set 2
CSE 373 Spring 2016
Due March 31 2016

1 Notes on Grading.

• You can write “I don’t know” for any question and receive 25% credit. You can take this option for any numbered problem, but not for part of a problem. For example, you can answer 3.1 and write “I don’t know” for 3.2, but you can’t write part of the solution for 3.2 and then write “I don’t know” for the rest.

• You get a 10% bonus for typing your homework. You are encouraged to use \textsc{E}\textsc{t}\textsc{e}X. You must type your entire homework to receive the bonus. The 10% bonus does not apply to problems answered with “I don’t know.”

2 Rotating Images

A gray-scale image of size \((n \times n)\) is a \((n \times n)\) matrix of integers. Given that we have a very quick algorithm \texttt{rectangularCopy()} to copy a rectangular chunk of pixel of size from one location to another:

1. Design a divide-and-conquer algorithm to rotate an image \(90^\circ\) clockwise. Assume that \(n\) is a power of 2. (You can use figures to help illustration).

2. If \(n\) is an arbitrary integer, how to modify the algorithm in question 1?

3. If the running time of \texttt{rectangularCopy()} on \((a \times a)\) matrix is \(O(a^2)\), what’s the running time of your algorithm?

4. If the running time of \texttt{rectangularCopy()} on \((a \times a)\) matrix is \(O(a)\), what’s the running time of your algorithm?

Grading:
1. 5 points for a correct an complete explanation of the algorithm (or pseudocode).
2. 5 points for generalizing to arbitrary sizes.
3. 5 points for each analysis. Must write down recurrence. Can use Master Theorem to solve it.

- **1. Solution:**

  Pseudocode of the algorithm is shown as following:

<table>
<thead>
<tr>
<th>Algorithm 1 Rotate Square Image 90° Clockwise</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: <strong>function</strong> RotateSq90Clockwise(A, n) ▷ Where A - input matrix, n - width of A(and power of 2)</td>
</tr>
<tr>
<td>2: if n ≤ 1 then</td>
</tr>
<tr>
<td>3: return A</td>
</tr>
<tr>
<td>4: else</td>
</tr>
<tr>
<td>5: A₁ ← RotateSq90Clockwise(A[(0 : (\frac{n}{2} - 1)), (0 : (\frac{n}{2} - 1)), (\frac{n}{2}))]</td>
</tr>
<tr>
<td>6: A₂ ← RotateSq90Clockwise(A[(0 : (\frac{n}{2} - 1)), ((\frac{n}{2}) : n - 1), (\frac{n}{2}))]</td>
</tr>
<tr>
<td>7: A₃ ← RotateSq90Clockwise(A[((\frac{n}{2}) : n - 1), ((\frac{n}{2}) : n - 1), (\frac{n}{2}))]</td>
</tr>
<tr>
<td>8: A₄ ← RotateSq90Clockwise(A[((\frac{n}{2}) : n - 1), (0 : (\frac{n}{2} - 1)), (\frac{n}{2}))]</td>
</tr>
<tr>
<td>9: A[(0 : (\frac{n}{2} - 1)), (0 : (\frac{n}{2} - 1))] ← rectangularCopy(A₄)</td>
</tr>
<tr>
<td>10: A[(0 : (\frac{n}{2} - 1)), ((\frac{n}{2}) : n - 1)] ← rectangularCopy(A₁)</td>
</tr>
<tr>
<td>11: A[((\frac{n}{2}) : n - 1), ((\frac{n}{2}) : n - 1)] ← rectangularCopy(A₂)</td>
</tr>
<tr>
<td>12: A[((\frac{n}{2}) : n - 1), (0 : (\frac{n}{2} - 1))] ← rectangularCopy(A₃)</td>
</tr>
<tr>
<td>13: return A</td>
</tr>
<tr>
<td>14: end if</td>
</tr>
<tr>
<td>15: end function</td>
</tr>
</tbody>
</table>

- **2. Solution:**

  (There are multiple possible approaches for this question. Any valid method will be considered correct in grading.)

Note that in the last question, we assume \(n\) is the power of 2 so that we don’t need to worry about the width of a matrix \(\frac{n}{2}\) not being integer in the recurrence. When \(n\) is not a power of 2, the only difficulty we have is to make sure that the size of matrix is an integer. To enable this, we define a function which rotate arbitrary size image as following:
Algorithm 2 Rotate Rectangular Image 90° Clockwise

1: function RotateRec90Clockwise(A, m, n) \(\triangleright\) Where A - input matrix, m - height(#row) of A, n - width(#column) of A
2: \hspace{1em} if \((m \leq 1) \land (n \leq 1)\) then
3: \hspace{2em} return A
4: \hspace{1em} else
5: \hspace{2em} A_1 \leftarrow \text{RotateRec90Clockwise}(A[0 : \lceil \frac{m}{2} \rceil - 1, 0 : \lceil \frac{n}{2} \rceil - 1], \lceil \frac{m}{2} \rceil, \lceil \frac{n}{2} \rceil)
6: \hspace{2em} A_2 \leftarrow \text{RotateRec90Clockwise}(A[0 : \lceil \frac{m}{2} \rceil - 1, (\lceil \frac{n}{2} \rceil : n - 1)], \lceil \frac{m}{2} \rceil, \lfloor \frac{n}{2} \rfloor)
7: \hspace{2em} A_3 \leftarrow \text{RotateRec90Clockwise}(A[(\lceil \frac{m}{2} \rceil : m - 1), (\lceil \frac{n}{2} \rceil : n - 1)], \lfloor \frac{m}{2} \rfloor, \lfloor \frac{n}{2} \rfloor)
8: \hspace{2em} A_4 \leftarrow \text{RotateRec90Clockwise}(A[(\lceil \frac{m}{2} \rceil : m - 1), (0 : \lceil \frac{n}{2} \rceil - 1)], \lfloor \frac{m}{2} \rfloor, \lceil \frac{n}{2} \rceil)
9: \hspace{2em} Create a matrix B of size \(n \times m\)
10: \hspace{2em} B[(0 : \lceil \frac{n}{2} \rceil - 1), (0 : \lfloor \frac{m}{2} \rfloor - 1)] \leftarrow \text{rectangularCopy}(A_4)
11: \hspace{2em} B[(0 : \lceil \frac{n}{2} \rceil - 1), (\lfloor \frac{m}{2} \rfloor : m - 1)] \leftarrow \text{rectangularCopy}(A_1)
12: \hspace{2em} B[(\lceil \frac{n}{2} \rceil : n - 1), (\lfloor \frac{m}{2} \rfloor : m - 1)] \leftarrow \text{rectangularCopy}(A_2)
13: \hspace{2em} B[(\lceil \frac{n}{2} \rceil : n - 1), (0 : \lceil \frac{m}{2} \rceil - 1)] \leftarrow \text{rectangularCopy}(A_3)
14: \hspace{2em} return B
15: end if
16: end function

For the case where \(n\) is not power of 2, function RotateRec90Clockwise(A, n, n) rotates the image 90° clockwise.

- 3. Solution:

Assuming \(f(a)\) is the running time of \textit{rectangularCopy()} on a \((a \times a)\) matrix, the recurrence is as following:

\[
T(n) = 4T\left(\frac{n}{2}\right) + 4f\left(\frac{n}{2}\right) + c' \\
= 4^2T\left(\frac{n}{2^2}\right) + 4^2f\left(\frac{n}{2^2}\right) + 4f\left(\frac{n}{2}\right) + c' \\
\vdots \\
= 4^iT\left(\frac{n}{2^i}\right) + 4^if\left(\frac{n}{2^i}\right) + 4^{i-1}T\left(\frac{n}{2^{i-1}}\right) + \ldots + 4f\left(\frac{n}{2}\right) + ic' \\
= 4^iT\left(\frac{n}{2^i}\right) + \sum_{j=1}^{i} 4^j f\left(\frac{n}{2^j}\right) + ic'
\]

The basis case is \(T(1) \leq c\), in which \(n/2^i = 1\). We have \(2^i = n\) and \(i = \log n\). Since we know that \(f(a) = O(a^2)\), it means \(\exists c_0 \text{ s.t. } f(a) \leq c_0(a^2)\). Furthermore we have:

\[
4^i f\left(\frac{n}{2^i}\right) \leq 4^i (c_0\left(\frac{n}{2^i}\right)^2) = c_0(4^i\left(\frac{n}{2^i}\right)^2) = c_0n^2
\]
Therefore we have

\[ T(n) = 4^i T\left(\frac{n}{2^i}\right) + \sum_{j=1}^{i} 4^j f\left(\frac{n}{2^j}\right) + ic' \]

\[ \leq cn^2 + \sum_{j=1}^{i} c_0 n^2 + ic' \]

\[ = cn^2 + \log n(c_0 n^2) + c' \log n \]

\[ = O(n^2 \log n) \]

4. Solution:

Similar to last question, we have

\[ T(n) = 4^i T\left(\frac{n}{2^i}\right) + \sum_{j=1}^{i} 4^j f\left(\frac{n}{2^j}\right) + ic' \]

The basis case is \( T(1) \leq c \), in which \( n/2^i = 1 \). We have \( 2^i = n \) and \( i = \log n \).

Since \( f(a) = O(a) \), it means \( \exists c_0 \) s.t. \( f(a) \leq c_0(a) \), and we have

\[ 4^i f\left(\frac{n}{2^i}\right) \leq 4^i (c_0\left(\frac{n}{2^i}\right)) = c_0(4^i\left(\frac{n}{2^i}\right)) = c_0(2^j n) \]

Therefore we have

\[ T(n) = 4^i T\left(\frac{n}{2^i}\right) + \sum_{j=1}^{i} 4^j f\left(\frac{n}{2^j}\right) + ic' \]

\[ \leq cn^2 + c_0 \sum_{j=1}^{i} (2^j n) + ic' \]

\[ = cn^2 + c_0(2^{i+1} - 2)n + ic' \]

\[ = cn^2 + c_0(2n - 2)n + c' \log n \]

\[ = (c + 2c_0)n^2 - 2c_0n + c' \log n \]

\[ = O(n^2) \]

3 Applications of findRankKElt

1. Show how to find all the \( k_1 \)- through \( k_2 \)-th-smallest elements of an array \( A \) of \( n \) distinct elements in \( O(n) \) time.
2. Show how to determine in linear time whether any element of an array \( A \) of \( n \) elements occurs at least \( n/2 \) times.

3. Given \( n \) distinct elements \( X = \{x_1, \ldots, x_n\} \) with weights \( w_1, \ldots, w_n \) such that \( \sum_i w_i = 1 \), the weighted median of \( X \) is the element \( x_k \) such that

\[
\sum_{i : x_i < x_k} w_i < \frac{1}{2}
\]

and

\[
\sum_{i : x_i > x_k} w_i \leq \frac{1}{2}
\]

Show how to compute the weighted median in linear time.

1. 5 points for this easy algorithm. Must be linear time for full credit. 1 point for an \( \omega(n) \) algorithm.

2. 5 points. Hashing not allowed. 2 points for the insight about an element that occurs \( n/2 \) times being the median, and 3 points for the rest of the algorithm.

3. 10 points. 5 points for realizing the algorithm needs to be generalized to more than just weighted median in order to be do-able as a divide and conquer. 5 points for the code. 1 point for an \( \omega(n) \) algorithm.

- **1. Solution:**
  - 1. \( a_1 \leftarrow \text{findRankKElt}(A, k_1) \)
  - 2. \( a_2 \leftarrow \text{findRankKElt}(A, k_2) \)
  - 3. Scan the array \( A \), return all elements \( x \) s.t. \( a_1 \leq x \leq a_2 \).

- **2. Solution:**
1. $a_{\text{midF}} \leftarrow \text{findRankKElt}(A, \left\lfloor \frac{n}{2} \right\rfloor)$
2. $a_{\text{midC}} \leftarrow \text{findRankKElt}(A, \left\lfloor \frac{n}{2} \right\rfloor + 1)$
3. Scan the array $A$, count the number($n_F$) of elements $x$ s.t. $x = a_{\text{midF}}$.
4. Scan the array $A$, count the number($n_C$) of elements $x$ s.t. $x = a_{\text{midC}}$.
5. If the number $n_F$ (or $n_C$) is larger than $\frac{n}{2}$, then there is an element ($a_{\text{midF}}$ or $a_{\text{midC}}$) occurs at least $\frac{n}{2}$ times. Otherwise there is no such element.

3. **Solution:**

(1). We define a function $(A', W') = \text{findAllElementBetweenK1nK2}(A, W, k_1, k_2)$. This function returns those elements in $A$ between the $k_1$th smallest and $k_2$th smallest elements($A'$), as well as the associated weights($W'$):

1. $a_1 \leftarrow \text{findRankKElt}(A, k_1)$
2. $a_2 \leftarrow \text{findRankKElt}(A, k_2)$
3. Scan the array $A$, return all elements $x$ s.t. $a_1 \leq x \leq a_2$ as $A'$. Meanwhile return all associated weights as $W'$.

Given that the size of $A$ is $n$, the complexity of findAllElementBetweenK1nK2() is $O(n)$.

(2). We define a function $(S) = \text{sumLargerThanKthElementsWeights}(A, W, k)$. This function find out in $A$ a set of elements larger than the $k$th smallest element, and sum up all associated weights:

1. $a_k \leftarrow \text{findRankKElt}(A, k)$
2. Scan the array $A$, find all elements $x$ s.t. $x \leq a_k$, and sum up all associated weights in $W$. Return the summation $S$.

The complexity of sumLargerThanKthElementsWeights() is $O(n)$ if the size of $A$ is $n$.

(3). An algorithm of finding the arbitrary weights split is as following:
Algorithm 3 Find Weights Split

1: function \textsc{FindWeightsSplit}(A, W, L, S_l, S_h, S_a) \triangleright Where A - input matrix, W - weights of A, L - size of A, S_l - lower-side weights summation, S_h - higher-side weights summation, S_a - all weights summation
2: \hspace{1em} \textbf{k} \leftarrow \frac{L}{2}
3: \hspace{1em} \textbf{x}^k \leftarrow \text{findRankKElt}(A, k)
4: \hspace{1em} \textbf{w}^k \leftarrow \text{weight of } \textbf{x}^k
5: \hspace{1em} \textbf{S}_h^k \leftarrow \text{sumLargerThanKthElementsWeights}(A, W, k)
6: \hspace{1em} \textbf{S}_l^k \leftarrow (S_a - S_h^k - w^k)
7: \hspace{1em} \textbf{if } \textbf{S}_l^k < S_l \land \textbf{S}_r^k \leq S_r \textbf{ then}
8: \hspace{2em} \text{return } \textbf{x}^k
9: \hspace{1em} \textbf{else if } \textbf{S}_l^k < S_l \land \textbf{S}_r^k > S_r \textbf{ then}
10: \hspace{2em} \textbf{S}'_l \leftarrow S_l - \textbf{S}_l^k
11: \hspace{2em} \textbf{S}'_h \leftarrow S_h
12: \hspace{2em} (A', W') \leftarrow \text{findAllElementBetweenK1nK2}(X, W, \frac{L}{2} + 1, L)
13: \hspace{2em} \textbf{L}' \leftarrow \text{length of } A'
14: \hspace{2em} \textbf{S}'_a \leftarrow \textbf{S}_h^k
15: \hspace{2em} x \leftarrow \text{FindWeightsSplit}(A', W', \textbf{L}', \textbf{S}'_l, \textbf{S}'_h, \textbf{S}'_a)
16: \hspace{2em} \text{return } x
17: \hspace{1em} \textbf{else if } \textbf{S}_l^k \geq S_l \land \textbf{S}_r^k \leq S_r \textbf{ then}
18: \hspace{2em} \textbf{S}'_h \leftarrow S_h - \textbf{S}_h^k
19: \hspace{2em} \textbf{S}'_l \leftarrow S_l
20: \hspace{2em} (A', W') \leftarrow \text{findAllElementBetweenK1nK2}(X, W, 1, \frac{L}{2} - 1)
21: \hspace{2em} \textbf{L}' \leftarrow \text{length of } A'
22: \hspace{2em} \textbf{S}'_a \leftarrow \textbf{S}'_l^k
23: \hspace{2em} x \leftarrow \text{FindWeightsSplit}(A', W', \textbf{L}', \textbf{S}'_l, \textbf{S}'_h, \textbf{S}'_a)
24: \hspace{2em} \text{return } x
25: \hspace{1em} \textbf{end if}
26: \textbf{end function}

\text{FindWeightsSplit}(A, W, L, \frac{1}{2}, \frac{1}{2}, 1) \text{ returns the weighted median as requested. Given that findAllElementBetweenK1nK2() and sumLargerThanKthElementsWeights() are linear}
time (of complexity $O(n)$), the complexity of FindWeightsSplit() is $T(n)$:

\[
T(n) \leq T\left(\frac{n}{2}\right) + c_1 n + c_2 n \\
\leq T\left(\frac{n}{2}\right) + c_3 n \\
\leq T\left(\frac{n}{4}\right) + c_3 (n + \frac{n}{2}) \\
\ldots \\
\leq T\left(\frac{n}{2^i}\right) + c_3 \sum_{j=0}^{i-1} \frac{n}{2^j} \\
\leq T\left(\frac{n}{2^i}\right) + c_3 (1 - \frac{1}{2^i}) n
\]

The basis case is $\frac{n}{2^i} = 1$ (or earlier), therefore,

\[
T(n) \leq c + c_3 (n - 1) = O(n)
\]

4 Finding the lonely element

Given a sorted array in which each number occurs twice except one number. Find the unique number using an order of less than $O(n)$ time. Hint: $O(\log n)$ time.

Example: \{1,1,2,3,3,5,5,8,8,13,13\}.

Solution:
We can identify some important properties:
- Since the array is sorted, all of repeated elements are consecutive.
- The size of the array is an odd number.
- If we try to divide the array as close as to half, the array will look like either \[1, 1, 2, 3\] or \[3, 4, 4, 5, 5\]
  or \[1, 1, 2, 3, 3\] \[4, 4, 5, 5\]

It can then be reasoned that the non repeating number is located in the odd-size part of the array. The binary search always divides the search space into two pieces. Dividing an odd size space gives two subspaces: one of the even size, and the second one of the odd size.

Divide the array arbitrarily, so that the first half is always even size. Then comparing the last element $L$ of the left subarray to the first element $R$ of the right subarray to find which half the extra element is located.
If $L \neq R$ then check the second half of the array, otherwise the left one.
Sample Code:

```cpp
def findLonelyElement(arr, size):
    if size == 1:
        return arr[0]

    medium = size // 2
    # make the size even number
    medium = medium % 2 == 0 ? medium : medium + 1;

    if arr == arr:
        # look in the left subarray
        return findLonelyElement(arr, medium - 1);
    else
        # look in the right subarray
        return findLonelyElement(arr + medium + 1, size - (medium + 1));
```

Since this is done through binary search, the time complexity is \(O(\log n)\).

5 A very frequent element

An array \(A[1..n]\) is said to have a dominant element if more than half of the entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a dominant element, and, if so, find that element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is \(A[i] > A[j]\)". (Think of the array elements as GIF files, say.) However, you can answer questions of the form "is \(A[i] = A[j]\)" in constant time.

**Solution:**

1. Show how to solve the problem in \(O(n \log n)\) time.
   For this problem, we'll apply the divide-and-conquer approach. Let \(A_1\) and \(A_2\) be the left and right halves of \(A\). Observe that if \(A\) has a dominant element \(x\), then \(x\)
must also be the dominant element of at least one of \(A_1\) or \(A_2\). We can recurse on \(A_1\) and \(A_2\), checking if either of the returned dominants represents more than half of \(A\):

\[
\text{function } n\log n\text{Dominant}(A) \\
\quad n = \text{len}(A) \\
\quad \text{if } n \leq 1 \text{ then} \quad \triangleright \text{base case: } n \text{ is } 0 \text{ or } 1 \\
\quad \quad \text{return } A \\
\quad \text{end if} \\
\quad \text{half} = n/2 \\
\quad A_1, A_2 = A[:\text{half}], A[\text{half:}] \quad \triangleright \text{split } A \text{ into halves} \\
\quad \text{maybeDominants} = n\log n\text{Dominant}(A_1) + n\log n\text{Dominant}(A_2) \quad \triangleright \text{concat returned lists of dominants (we could optimize by checking these separately)} \\
\quad \text{for } \text{md} \text{ in maybeDominants do} \\
\quad \quad \text{numOccurences} = \text{sum}(1 \text{ for } ai \text{ in } A \text{ if } ai == \text{md}) \quad \triangleright \text{number of times } \text{md} \text{ occurs in } A \\
\quad \quad \text{if } \text{numOccurences} > \text{half} \text{ then} \\
\quad \quad \quad \text{return } [\text{md}] \\
\quad \text{end if} \\
\quad \text{end for} \\
\quad \text{return } [] \quad \triangleright \text{A has no dominant element} \\
\text{end function}
\]

**Time:** Every nonterminating call to \(n\log n\text{Dominant}\) results in 2 recursive calls on lists of half the size. This means that every level of our call tree has all \(n\) elements, split more or less evenly amongst the nodes. Because we terminate when \(A\) has 0 or 1 element(s), we know that our tree will have \(\Theta(\log_2 n)\) levels. In the worst case, each recursive call returns a dominant, so each parent has to scan its entire portion of \(A\), meaning that each level has to cumulatively scan all of \(A\) in \(O(n)\) time. We do \(O(n)\) work \(\Theta(\log n)\) times, so our overall time complexity is \(O(n \log n)\).

**Space:** The above pseudocode may take \(O(n \log n)\) space if \(A_1\) and \(A_2\) are being copied from \(A\), but by operating on \(A\) in memory we can achieve \(O(\log n)\) space. We can even whittle it down to \(O(1)\) space by formulating the procedure as a stack-free loop. This is possible because we can infer the parameters of the parent and siblings from those of the child, meaning we don’t need any stack to remember them for us. Again, this assumes that our counters/pointers can go up to the length of \(A\).

2. Can you give a linear time algorithm?
Yes. See the Boyer-Moore majority vote algorithm.
6 Tetromino cover

Is it possible to cover a chessboard with 15 T-tetrominoes and a square one?

Solution
No, it is NOT possible to cover a chessboard with 15 T-tetrominoes and a square one.

Facts:

- A chessboard has an even number of white and black squares.
- A T-tetromino can cover either (3 white and 1 black squares) or (3 black and 1 white squares). We can also put the statement in this way: a T-tetromino can cover an odd number of white and an odd number of black squares.
- A square tetromino can cover either (2 white and 2 black squares) or (2 black and 2 white squares).
- An even number + an odd number = an odd number
- An odd number × an odd number = an odd number

1. In order to cover the chessboard, the number of white squares and the number of black squares that 15 T-tetrominoes and a square one cover have to be even. (To obey the first bullet).

2. We have 15 T-tetrominoes. No matter how we place them. They will cover an odd number of white and an odd number of black squares.

3. Second statement is true because: a T-tetromino can cover an odd number of white and an odd number of black squares ×15 T-tetrominoes = an odd number of white and an odd number of black squares we can cover. (Using last bullet)

4. After placing 15 T-tetrominoes, we can place a square tetromino to cover two more white and black squares.

5. Using fourth bullet, the numbers of white squares and the number of black squares we covered are both odd numbers. This violates the first statement.

6. Therefore, it is NOT possible to cover a chessboard with 15 T-tetrominoes and a square one.
7 Infinite Array

Suppose there is an array with infinitely expandable number of elements. The first \( n \) elements are integers in sorted order, while the rest of the array is filled with \( \infty \) sign. The value of \( n \) is not known.

Describe an algorithm that returns the position of an integer \( x \) if it exists in the array, and returns null if \( x \) does not exists. The algorithm must take \( O(\log n) \) time. (You can assume that the implementation of such array returns an error message \( \infty \) to you when elements \( A[i] \) with \( i > n \) are accessed.)

**Solution** Let the infinite array \( A[.] \) start with index 1. Given an integer \( x \) as input, we will do \( F(1,A,x) \) to find the position of \( x \) in \( A \). If \( x \) is not found then it will return -1. The general approach is to double the increment of the searching index to find the position of \( x \).

The function \( F(i, A, x) \) where \( i \) is the index to start, \( A \) is the array to search, and \( x \) is the integer to find in the array.

**Base case:**
if \( x = A[i] \),
    then return \( i \).
if \( x < A[i] \) or \( A[i] \) is infinity
    then return -1.

Otherwise (when \( x > A[i] \)):
    For \( i = i \times 2 \) (in other words \( i \) doubles every time).
        If \( A[i] \) is an infinity (giving error message)
            then \( F(i/2, A, x) \)
        Else if \( x = A[i] \)
            then return \( i \)
        Else if \( x > A[i] \)
            then continue to the next \( i \).
        Else if \( x < A[i] \)
            then \( F(i/2, A, x) \)

This would be done in \( O(\log n) \) time because it increments the searching index by double.

Another Approach:
Find the index of the end of valid elements, which is \( n \), by using idea similar to binary search (keeping variables left and right, doubling the increment)

After getting \( n \), just apply binary search on \( A[1...n] \). Done. Analysis: Find \( n \) by exponentiation takes \( O(\log n) \). Binary search is \( O(\log n) \). So, total time is \( O(\log n) \).
import math

def infinite_array(A, x):
    if A[0] > x:
        return -1
    if A[0] == x:
        return 0

    left, right = 1, 1
    while A[right] != "INF":
        left = right
        right = 2 * right

    mid = 0
    while (right-left) > 1:
        mid = left + int((right-left)/2)
        if A[mid] == "INF":
            right = mid
        else:
            left = mid

    return binary_search(A[left:right], x)

def binary_search(A, x):
    if len(A) == 0:
        return -1
    left, right = 0, len(A)
    while (left < right):
        mid = left + int((right-left)/2.0)
        if A[mid] == x:
            return mid
        elif A[mid] < x:
            left = mid + 1
        else:
            right = mid
    return -1

A = [1,3,5,6,7,8,9,11,14,19,1000, 113123, 1125122, 10000000, "INF", "INF", "INF", "INF", "INF", "INF", "INF", "INF", "INF", "INF", "INF", "INF", "INF", "INF", "INF", "INF", "INF", "INF"]
print infinite_array(A, 13122313)
print infinite_array(A, 14)