Section II.6: Analysis of Iterative Searches and Sorts

Binary Search

Binary Search is an efficient algorithm that searches for an item $X$ in a sorted array. The item $X$ searched for is compared with the value of the midpoint of the list. If $X$ is found, then the position in the list is given. If $X$ is less than this value, we search the left half of the list (excluding the midpoint). If $X$ is greater than this value, we search the right half of the list (excluding the midpoint). This process is continued until the length of the sublist is 1. If $X$ is found, the position in the list is returned. If $X$ is not found, then the position 0 is returned.

Iterative pseudo code for Binary Search, found in Epp, p. 521 is given below.

Algorithm Binary Search

Input: positive integer $n$, array $A[1], A[2], \ldots, A[n]$ in ascending order, item $X$ of the same data type as the elements of the array.

Output: 0 (zero) if $X$ is not found, otherwise the position in the array where $X$ is found.

Algorithm Body:

1. $index := 0$;
2. $bot := 1$;
3. $top := n$;
4. while $(bot \leq top$ and $index = 0)$
   - $mid := \left\lfloor \frac{bot + top}{2} \right\rfloor$
   - if $X = A[mid]$ then $index := mid$
   - else if $X < A[mid]$ then $top := mid - 1$
   - else [X > A[mid]] $bot := mid + 1$;
5. return $index$

end Algorithm Binary Search

Example II.6.1:
(a) Trace the action of Binary Search on the list 1, 8, 10, 12, 17, 19, 23 if we are searching for item 10.
(b) Trace the action of Binary Search on the list 1, 8, 10, 12, 17, 19, 23 if we are searching for item 15.

Solution: (a) Iteration Number

<table>
<thead>
<tr>
<th>Iteration Number</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$mid$</td>
<td>--</td>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>$X \ ? A[mid]$</td>
<td>--</td>
<td>&lt;</td>
<td>&gt;</td>
<td>=</td>
</tr>
<tr>
<td>$bot$</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$top$</td>
<td>7</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>$index$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

The item is found in position 3.
**Solution:** (b) Iteration Number 0 1 2 3

<table>
<thead>
<tr>
<th>Variable</th>
<th>mid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Names</td>
<td>X ? A[mid]</td>
</tr>
</tbody>
</table>

| bot      | 1 5 5 5 |
| top      | 7 7 5 4 |
| index    | 0 0 0 0 |

The item is not found, *index* 0 is returned.

Tree Structure for Binary Search

**Definitions (Informal):** A tree is a data structure such that each node has at most one predecessor (parent) node. A node may have many successor (children) nodes. Trees are formally covered later but are informally used from now on for illustrative purposes. There is one node without any parent (the root), and the leaves are nodes that have no children. The level of a node is the length of the path from the root to that node. The root has level 0. A Binary Tree is one where each node has at most two children.

The binary tree for Binary Search is constructed as follows. At each level the midpoint of that sublist is the parent and the midpoints of the left and right sublists are the children. The root of the tree is the midpoint of the original list. The leaves are the nodes for which the left and right sublists are empty.

**Example II.6.2 (a):** Draw the tree for the list in Example 1.

```
        4-12
       /   \
      2-8  6-19
     /   \
   1-1 3-10 5-17 7-23
```

The solution is shown above. The first number in each node is the position of the item in the list, and the second is the item itself.

**Example II.6.2 (b):** Draw the tree for list 1, 4, 7, 8, 10, 12, 15, 17, 19, 22, 23.

```
        6-12
       /   \
      3-7  9-19
     /   \
   1-1  4-8 7-15 10-22
   /   \
  2-4 5-10 8-17 11-23
```
The solution is shown above. The first number in each node is the position of the item in the list, and the second is the item itself.

**Analysis of Iterative Binary Search**

We wish to find (a) the run time \( T(n) \) and (b) a function \( U(n) \) with one term, and leading coefficient 1, where \( T(n) = \Theta(U(n)) \).

**Solution: (a)**

**Case 1:** If the size of the array is \( n = 2^k - 1 \) for some positive integer \( k \), then the tree structure is full, i.e., each level has the maximum number of nodes. (Example 1 is such a case for \( k = 3 \)). The maximum number of subdivisions of the original array is the maximum number of nodes from the root to the leaf level of the tree and this value is \( k \). Since \( 2^k=n+1 \), we have \( k = \log_2(n+1) = \lceil \log_2(n+1) \rceil \) since \( \log_2(n+1) \) is an integer. Therefore the run time \( T(n) = \log_2(n+1) \).

**Case 2:** \( 2^k \leq n < 2^{k+1} - 1 \) for some positive integer \( k \). The tree for binary search requires at least one node at the next level of the tree. So the maximal length of a path through the tree is \( \lceil \log_2(n+1) \rceil \). We must use the ceiling function in this case since \( \log_2(n+1) \) is not an integer. Therefore the run time \( T(n) = \log_2(n+1) \).

In either case, then, \( T(n) = \lceil \log_2(n+1) \rceil \).

**Solution: (b)** We show \( T(n) = \Theta(\log_2 n) \).

To show \( T(n) = O(\log_2 n) \), note that \( \log_2(n+1) \leq \log_2 n \leq 2 \log_2 n \) for all integers \( n \geq 2 \). In the definition of big O, we can take \( C = 2, \ X_0 = 2 \). In the above inequality we use \( 1 \leq \log_2 n \) and fact that the slope of the graph of \( \log_2(n+1) \) is less than the slope of the graph of \( \log_2 n + 1 \). We also use the fact that \( \log_2(n+1) = \log_2 n + 1 \) for \( n = 1 \).

To show \( \log_2 n = O(\lceil \log_2 (n+1) \rceil) \), note that \( \log_2 n < \log_2 (n+1) \leq \lceil \log_2 (n+1) \rceil \).

So \( C = 1 \) and \( X_0 = 1 \) from the definition of big O can be used.

**Insertion Sort**

Insertion Sort is a sort that creates from the input array \( A \) of size \( n \), \( n \) a positive integer, a sorted subarray of size \( i \) for each \( i, 1 \leq i \leq n \). Once a sorted array of size \( i-1 \) is obtained, for \( 2 \leq i \leq n \), we assign \( A[i] \) to \( v \). \( v \) is compared to \( A[i-1] \). If \( v < A[i-1] \) then one copies \( A[i-1] \) into \( A[i] \). Then \( v \) is compared to \( A[i-2] \). If \( v < A[i-2] \) then the elements \( A[i-2] \) is moved up. This process continues until \( v \geq A[j] \) for some \( j, 0 \leq j \leq i \). If the condition \( v < A[1] \) is reached, then we must compare \( v \) to the sentinel \( A[0] \). One obtains \( v > A[0] \), and this causes the while loop to terminate. This process is repeated for \( i = 2, i = 3, \ldots \) up to \( i = n \). Finally we get a sorted array of size \( n \).
The pseudo code for Insertion Sort is as follows:

**Algorithm Insertion Sort**

**Input:** Positive integer \( n \), array \( A \) of \( n \) items.

**Output:** Sorted array in ascending order.

**Body of algorithm:**

\[
\begin{align*}
A[0] &:= \text{sentinel [ put a value less than all others in the list into } A[0]] \\
\text{for } i &:= 2 \text{ to } n \\
v &:= A[i] \\
j &:= i \\
\text{while } A[j-1] > v \\
j &:= j - 1 \\
\end{align*}
\]

\( \text{end while; } \)

\( A[j] := v; \)

\( \text{next } i; \)

**End Algorithm Insertion Sort**

**Analysis of Insertion Sort**

**Worst Case:** The list is in descending order. For \( i \) between 2 and \( n \), there are \( i \) comparisons when \( A[i] \) is smaller than all previous values. The number of comparisons, including a comparison against the sentinel, is

\[
T(n) = \sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1 = \frac{n^2 + n - 2}{2} = \Theta(n^2).
\]

**Best Case:** The list is already sorted. When \( A[i+1] \) is compared with \( A[i] \), it is always larger. So only one comparison is made for each \( i \). The number of comparisons is

\( T(n) = n - 2 + 1 = n - 1 = \Theta(n). \)

**Average Case:** Assume that in any subarray of \( A \) of length \( i \), the elements are uniformly distributed, i.e., the maximum element is equally likely to be in any position. Let \( C \) be the total number of comparisons needed, and \( C_i \) be the number of comparisons needed when we insert the element at index \( i \) in its proper position in the subarray \( A[1], \ldots, A[i] \) where \( A[1], \ldots, A[i-1] \) is already sorted. Then

\[
T(n) = E[C] = E[C_2 + \ldots + C_n] = \sum_{i=2}^{n} E[C_i]
\]

by the linearity of expectation (Theorem I.2.1). We now find \( E[C_i] \). Let \( P_i(k) \) be the probability that \( \max \{ A[1], \ldots, A[i] \} \) is in the \( k \)-th position, \( 1 \leq k \leq i \). Then

\[
P_i(k) = \frac{1}{i}.
\]
Let $C_i(k)$ be the number of comparisons needed if the element at index $i$ is to be put in index $k$, where the subarray $A[1], \ldots, A[i-1]$ is already sorted. Then $C_i(k) = i - k + 1$. We then have

$$E[C_i] = \sum_{k=1}^{i} P_i(k) \cdot C_i(k) = \sum_{k=1}^{i} \frac{1}{i} \cdot (i - k + 1) = \frac{1}{i} \sum_{j=1}^{i} j = \frac{1}{i} \cdot \frac{(i + 1) \cdot i}{2} = \frac{i + 1}{2}$$

Therefore

$$T(n) = \sum_{i=2}^{n} E[C_i] = \sum_{i=2}^{n} \frac{i + 1}{2} = \frac{1}{2} \cdot \frac{(n+1)(n+2)}{2} - \frac{1}{2} \cdot (1 + 2)$$

$$= \frac{n^2 + 3n + 2 - 6}{4} = \frac{n^2 + 3n - 4}{4} = \Theta(n^2).$$

Note that the average run time is about half that of the worst case.

**Example II.6.3:** Given an array $A$ that is 4, 3, 1, 9, 6

(a) Give the arrangement of the array $A$ at the end of each iteration of the for loop in the Insertion Sort algorithm.

(b) Compute the total number of comparisons needed.

(c) Compute the number of comparisons needed in the best case, the worst case, and the average case.

**Solution:**

(a) Initially:

<table>
<thead>
<tr>
<th>value of $i$</th>
<th>array $A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3 4 1 9 6</td>
</tr>
<tr>
<td>3</td>
<td>1 3 4 9 6</td>
</tr>
<tr>
<td>4</td>
<td>1 3 4 9 6</td>
</tr>
<tr>
<td>5</td>
<td>1 3 4 6 9</td>
</tr>
</tbody>
</table>

(b) value of $i$

<table>
<thead>
<tr>
<th>Number of Comparisons</th>
<th>2 3 4 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total</td>
<td>2 3 1 2</td>
</tr>
</tbody>
</table>

Note that if $A[i] < A[1]$, then $A[i]$ must be compared to the sentinel $A[0]$. Therefore, we have 2 comparisons for $i = 2$ and 3 comparisons for $i = 3$.

(c) Plug in 5 for $n$ in above formulas for the best case, the worst case, and the average case.

Best case: $T(n) = 5 - 1 = 4$

Worst Case: $T(n) = \frac{5^2 + 5 - 2}{2} = 14$

Average case: $T(n) = \frac{5^2 + 3 \cdot 5 - 4}{4} = \frac{36}{4} = 9$
Exercises:

(1) Let $A$ be the sorted array $1, 7, 9, 12, 17, 21, 30$. Use the format of Example II.6.1, parts (a) and (b).
(a) Fill in the trace table for Binary Search when $X = 9$ is searched for.
(b) Fill in the trace table for Binary Search when $X = 31$ is searched for.
(c) Draw a tree structure for this list (see Example II.6.2).

(2) Let $A$ be the sorted array $1, 3, 4, 5, 9, 22, 24, 31, 35, 36, 39, 41$. Use the format of Example II.6.1, parts (a) and (b).
(a) Fill in the trace table for Binary Search when $X = 39$ is searched for.
(b) Fill in the trace table for Binary Search when $X = 8$ is searched for.
(c) Draw a tree structure for this list (see Example II.6.2).

(3) Compute the best case, the average case, and the worst case for the run time $T(n)$ of insertion sort for an array of 16 elements.

(4) Let $A$ be an array of elements $5, 8, 2, 13, 4, 1, 15, 9$. (a) Using the method of example II.6.3, give the arrangement of array $A$ at the end of each iteration of the for loop in the Insertion Sort algorithm. (b) Compute the actual number of comparisons of $A[i]$ to previous elements for each $i$ between 2 and 8. (c) Get the total number of comparisons needed.