1. An algorithm is a finite set of precise instructions for performing a computation.

2. Properties that algorithms generally share:
   \textit{Input, Output, Definiteness, Correctness, Effectiveness, Generality.}

3. Fundamental constructs: \textit{if-then; while; for loops.}

4. Example: \textsc{Find-Max} (Algorithm 1).

5. Two Search Algorithms:
   \begin{itemize}
   \item (a) Linear search (Algorithm 2).
   \item (b) Binary search (Algorithm 3).
   \end{itemize}

6. Two Sorting Algorithms:
   \begin{itemize}
   \item (a) Bubble sort (Algorithm 4).
   \item (b) Insertion sort (Algorithm 5).
   \end{itemize}

7. Greedy algorithms (make the best choice at each step with only knowledge available).

8. Complexity of algorithms: measured in space (memory) and time requirements.
   Time complexity is expressed in terms of the number of operations used by the algorithm when the input has a particular size.

   \begin{itemize}
   \item (a) Time complexity of \textsc{Linear-Search}:
     If $x = a_i$ for some $i$, then $2i + 1$ comparisons. Therefore, the most comparisons are $2n + 2$ (= $2n$ (in the loop) + 1 (exit loop) + 1 (outside loop)), are required when the element is not in the list.
   \item (b) Time complexity of \textsc{Binary-Search}:
     Assume that there are $n = 2^k$ elements in the list. At most $2k + 1 + 1 = 2 \log n + 2$ comparisons are required where $2k + 1 + 1 = 2 \times$ (# of loops) + 1 (exit loop) + 1 (outside loop).
   \item (c) Time complexity of \textsc{Bubble-Sort}:
     The total number of comparisons is $(n - 1) + (n - 2) + \cdots + 2 + 1 = \frac{(n-1)n}{2}$.
   \item (d) Time complexity of \textsc{Insertion-Sort}:
     The total number of comparisons is $1 + 2 + \cdots + (n - 1) = \frac{(n-1)n}{2}$.
   \end{itemize}
Algorithm 1 An algorithm for finding the maximum element in a finite sequence $a = \{a_1, \ldots, a_n\}$ of integers.

**FIND-MAX** $(a)$

```
max ← a_1
for $i ← 2, 3, \ldots, n$ do
    if $max < a_i$ then
        $max ← a_i$
    end if
end for
return $max$ \{ $max$ is the largest element. \}
```

Algorithm 2 An algorithm for locating an element $x$ in a list of distinct elements $a = \{a_1, \ldots, a_n\}$ of integers, or determine that it is not in the list.

**LINEAR-SEARCH** $(a, x)$

```
i ← 1
while $i \leq n$ and $x \neq a_i$ do
    $i ← i + 1$
end while
if $i \leq n$ then
    location ← $i$
else
    location ← 0
end if
return $location$ \{ $location$ is the subscript of the term that equals $x$, or is 0 if $x$ is not found. \}
Algorithm 3 An algorithm for locating an element $x$ in a list of integers $a = \{a_1, \ldots, a_n\}$ occurring in an increasing order, or determine that it is not in the list.

**BINARY-SEARCH** $(a, x)$

1. $i \leftarrow 1$ \{ $i$ is the left endpoint of search interval \}
2. $j \leftarrow n$ \{ $j$ is the right endpoint of search interval \}
3. while $i < j$ do
   4. \hspace{1em} $m \leftarrow \lfloor (i + j)/2 \rfloor$
   5. \hspace{1em} if $x > a_m$ then
   6. \hspace{2em} $i \leftarrow m + 1$
   7. \hspace{1em} else
   8. \hspace{2em} $j \leftarrow m$
   9. \hspace{1em} end if
10. end while
11. if $x = a_i$ then
12. \hspace{1em} location $\leftarrow i$
13. else
14. \hspace{1em} location $\leftarrow 0$
15. end if
16. return location \{ location is the subscript of the term that equals $x$, or is 0 if $x$ is not found \}

Algorithm 4 An algorithm for sorting a list of integers $a = \{a_1, \ldots, a_n\}$ into a list in which the elements are in increasing order.

**BUBBLE-SORT** $(a)$

1. for $i \leftarrow 1, \ldots, n - 1$ do
2. \hspace{1em} for $j \leftarrow 1, \ldots, n - i$ do
3. \hspace{2em} if $a_j > a_{j+1}$ then
4. \hspace{3em} temp $\leftarrow a_j$
5. \hspace{3em} $a_j \leftarrow a_{j+1}$
6. \hspace{3em} $a_{j+1} \leftarrow temp$
7. \hspace{2em} end if
8. \hspace{1em} end for
9. \hspace{1em} end for
10. return $a$ \{ $a_1, a_2, \ldots, a_n$ is in increasing order \}
Algorithm 5 An algorithm for sorting a list of integers $a = \{a_1, \ldots, a_n\}$ into a list in which the elements are in increasing order.

\begin{algorithm}
\textsc{Insertion-Sort} ($a$) \\
\hspace{1em} \textbf{for} $j \leftarrow 2, \ldots, n$ \textbf{do} \\
\hspace{2em} $i \leftarrow 1$ \\
\hspace{2em} \textbf{while} $a_j > a_i$ \textbf{do} \\
\hspace{3em} $i \leftarrow i + 1$ \\
\hspace{2em} \textbf{end while} \\
\hspace{2em} $m \leftarrow a_j$ \\
\hspace{2em} \textbf{for} $k \leftarrow 0, \ldots, j - i - 1$ \textbf{do} \\
\hspace{3em} $a_{j-k} \leftarrow a_{j-k-1}$ \\
\hspace{2em} \textbf{end for} \\
\hspace{2em} $a_i \leftarrow m$ \\
\hspace{1em} \textbf{end for} \\
\hspace{1em} \textbf{return} $a \{a_1, a_2, \ldots, a_n\}$ is in increasing order
\end{algorithm}

The HALTING Problem, 1936 Alan Turing. One of the most famous theorems in computer science. There exist problems that cannot be solved using any procedure (i.e. there exist unsolvable problems). There does not exist a procedure which given a program $P$ and input to the program $I$ determines whether program $P$ completes (halts) given input $I$ or runs forever. A subtle proof by contradiction: