The Process of Algorithmic Analysis

Comp 363 Fall Semester 2003

September 1, 2003

The purpose of these notes is to supplement Sections 2.1 and 2.2 of the textbook.

1. Problems

A problem \( P \) that is to be subjected to the process of algorithmic analysis must have the following characteristics:

- \( P \) must be suitable for solution on a digital computer and
- the set of instances of \( P \) must decompose into sets of size \( n \), \( n \geq 1 \).

Any problem that manipulates a finite set of objects is generally amenable to solution by computer. Usually, but not always, the measure of size to be used in the analysis of the problem \( P \) is obvious from the description of \( P \).

Example 1.1. Let \( P = \text{Towers of Hanoi} \). In this case, the set of objects is the set of disks, and an instance of size \( n \) is a \( n \)-disk problem.

Example 1.2. Let \( P = \text{Sorting} \). Here, the set of objects is the set of list items to be sorted, and an instance of size \( n \) is a list of \( n \) items to be sorted.

For any problem \( P \), let \( I_n \) be the set of all instances of \( P \) of size \( n \). Note that if \( P = \text{Towers of Hanoi} \), then \( |I_n| = 1 \) since there is only one \( n \)-disk problem. On the other hand, if \( P = \text{Sorting} \), then \( |I_n| = \infty \) since there are infinitely many lists of size \( n \).
2. Models

Given a problem \( P \), a model for \( P \) is a set \( M \) of legal operations that can be used to design algorithms for solving \( P \).

Example 2.1. Let \( P = \text{Crossing the Rocky Mountains} \). Let

\[
M = \left\{ \begin{array}{c}
\text{walk in hiking boots, walk in snow shoes,} \\
\text{use cross-country skis, use a sled,} \\
\text{use mountain-climbing equipment}
\end{array} \right. 
\]

Clearly, flying across the mountains in an airplane is a feasible way of crossing the Rockies. However, it is an illegal operation within the model \( M \). Only those methods of transportation listed in \( M \) can be used to design an itinerary across the mountains.

Example 2.2. Let \( P = \text{Towers of Hanoi} \). Let

\[
M = \left\{ \begin{array}{c}
\text{moving a disk subject to the rules,} \\
\text{comparing integers, executing an} \\
\text{if-then-else operation, implementing} \\
\text{recursion, etc.}
\end{array} \right. 
\]

Once a model for a problem has been established, algorithms for solving the problem are designed using only those operations within the chosen model. To measure the amount of time required by the algorithm, simply count the number of legal operations used by the algorithm.

Is this approach reasonable?

This question is hard to answer. But consider the following line of reasoning. Suppose an algorithm for some problem has been designed within a given model. To run the algorithm on a computer, certain choices must be made. For example,

- a high-level language must be chosen in which to code the algorithm and
- a compiler must be chosen (usually with a certain computer architecture in mind) to translate the statements in the high-level language into machine language instructions.
It seems reasonable to assume that each operation in the model will translate into at most a constant number of statements in the high-level language, and in turn, each statement in the high-level language will translate into at most a constant number of machine instructions. Each machine instruction requires at most a constant number of machine clock periods to execute. (A clock period is a fixed, very small time interval during which the computer hardware performs the most primitive types of operations. Typically, clock periods are measured in tens of nanoseconds.) Therefore, the total number of clock periods needed to run the algorithm is certainly bounded above by a constant multiple of the total number of legal operations used by the algorithm. That is,

\[
\left( \frac{\text{total number of machine clock periods needed to run the algorithm}}{\text{total number of legal operations needed to execute the algorithm}} \right) \leq k \cdot \left( \frac{\text{total number of legal operations needed to execute the algorithm}}{\text{total number of legal operations needed to execute the algorithm}} \right)
\]

where \( k \) is some positive constant. Note that this inequality relates time and operations. Therefore, counting the number of legal operations at least gives us a quantity that is proportional to the total running time of the algorithm.

### 2.1. Basic Operations

Typically, a given model \( M \) for a problem has many legal operations in it making it hard to obtain an accurate count of the total number of operations used. Therefore, in practice, only a few (one or two at most) of the operations in \( M \) are selected to be actually counted. The selected operations are called basic operations. Some authors call them essential operations or even barometer operations.

**Example 2.3.** Let \( P = \text{Towers of Hanoi} \). In this case, let the set of basic operations be \{move a disk\}. Notice that we have simply discarded the other legal operations.

**Example 2.4.** Let \( P = \text{Sorting} \). Here, we might let the set of basic operations be \{compare two list items, swap the positions of two list items\}.

The set of basic operations must be chosen so that the number of basic operations performed by the algorithm is proportional to the number of legal operations used by the algorithm. In this case, the inequality above will still hold for some choice of constant \( k \). We have to be careful when we decide which of the legal operations should be considered basic operations. For example, suppose we are planning to hike across the Rocky Mountains. Suppose also that it is Summer,
and that we choose our route through lower elevations so that no snow will be encountered. After we have chosen our route, we then decide that the basic operation will be ‘using cross-country skis’. Since no snow will be encountered, our route will require no work! Obviously, we have cheated here! Some other legal operation must be chosen to be the basic one.

**Example 2.5.** Let \( P = \text{Towers of Hanoi} \). If the basic operation is moving a disk, then method \( \text{hanoi} \) presented in class requires \( 2^n - 1 \) operations on an \( n \)-disk problem.

### 2.2. Running Time

Let \( A \) be an algorithm for solving a problem \( P \) within some model \( M \). If \( I \) is some instance of \( P \) of size \( n \) (that is, \( I \in I_n \)), define

\[
    t_A(I) = \begin{cases} 
        \text{number of basic operations} \\
        \text{needed by algorithm } A \text{ to} \\
        \text{process the instance } I 
    \end{cases}.
\]

Intuitively, the quantity \( t_A(I) \) represents the running time of the algorithm on the instance \( I \). Certainly, as explained above, it is proportional to the actual running time (measured in clock periods on some computer) of the algorithm \( A \) on the instance \( I \).

**Example 2.6.** Let \( P = \text{Towers of Hanoi} \). If \( A \) is method \( \text{hanoi} \) and \( I \) is the (unique) instance of \( P \) of size \( n \), then \( t_A(I) = 2^n - 1 \). Therefore, the quantity \( 2^n - 1 \) is proportional to the running time of \( \text{hanoi} \) (in clock periods) on any computer.

### 3. Worst Costs

There are two notions of worst cost, the *worst cost of an algorithm* and the *worst cost of a problem*. They are different concepts, and it is critical to understand the difference!
3.1. Worst Cost of an Algorithm

Let $A$ be an algorithm for solving a problem $P$ within a model $M$. The \textit{worst cost of $A$ on inputs of size $n$} is defined as

$$c_A(n) = \max_{I \in I_n} \{t_A(I)\}.$$  

Therefore, $c_A(n)$ is the largest number of basic operations needed by the algorithm to process any instance of size $n$. The number $c_A(n)$ is a measure of the \textit{worst-case performance} of the algorithm $A$.

**Example 3.1.** Let $P =$ Towers of Hanoi. If $A$ is method \texttt{hanoi}, $c_A(n) = 2^n - 1$.

**Example 3.2.** Let $P =$ Searching an Unordered List, and suppose $A$ is linear search. Specifically, $A$ simply scans the list of items starting with the first item until it either finds the one it is looking for or scans the entire list without finding the item. Clearly, $c_A(n) = n$ since $A$ might have to compare the search item with every item on the list.

3.2. Worst Cost of a Problem

Let $P$ be a problem, and let $M$ be a model for $P$. The \textit{worst cost of $P$ on inputs of size $n$} is defined as

$$c_P(n) = \min_A \{c_A(n)\}$$

where the minimum is taken over all algorithms $A$ for $P$ designed within the model $M$. Intuitively, any algorithm constructed to solve problem $P$ \textit{must perform at least $c_P(n)$ basic operations in the worst case}. It is hard (if not impossible!) to compute $c_P(n)$ exactly. Notice that if $A$ is any algorithm for $P$, $c_P(n) \leq c_A(n)$.

That is, the worst-cost performance of any algorithm $A$ for problem $P$ yields an upper bound on the worst cost of $P$.

**Example 3.3.** Let $P =$ Towers of Hanoi. By a result proved in class, $c_A(n) \geq 2^n - 1$ for any algorithm $A$ that solves the Towers of Hanoi problem. Therefore, by definition, $c_P(n) \geq 2^n - 1$. Actually, we have proved that $c_P(n) = 2^n - 1$ since method \texttt{hanoi} uses exactly $2^n - 1$ moves to solve the problem. The Towers of Hanoi problem is one of the rare instances in which $c_P(n)$ can be computed exactly.