CSE 101 - ALGORITHMS - SUMMER 2000
Lecture Notes 1

Monday, July 3, 2000.

Chapter 1

Introduction to Algorithms

Algorithm. An **algorithm** is a mechanical or computational procedure that takes an **input** and produces an **output**. An input or an output is usually a structure containing one value or a collection of values grouped under some well-established conventions. An algorithm *transforms* the input into the output, by performing a sequence of computational **steps**. During its execution, an algorithm goes through a series of **states**.

Computational Problem. We can view an algorithm as a tool for solving a **computational problem**, which is a relationship between an input and an output specified in general terms, sometimes just plain English. Sometimes, the input and the output are clearly specified like in the following well-known sorting problem:

SORTING

INPUT: A sequence A of n numbers $\langle a_1, a_2, ..., a_n \rangle$.

OUTPUT: A permutation (reordering) $\langle a'_1, a'_2, ..., a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots a'_n$.

Often, the input and the output are understood from the context, like in the following problem:

PRIME

Given a number q, is q prime?

where the input consists of a natural number q and the output consists of either YES or NO.

Instance. An **instance** or **input** of a problem is a value for its input. For example, $\langle 17, 21, 8, 3, 15, 21, 9, 7 \rangle$ is an instance of **SORTING**, while 29 is an instance of **PRIME**. An algorithm is **correct** for a certain problem if and only if it terminates on each input of that problem and returns the desired output. Thus, a correct algorithm for **SORTING** would terminate for any sequence of natural numbers as input and would return the sorted sequence, $\langle 3, 7, 8, 9, 15, 17, 21, 21 \rangle$ for the instance above. Similarly, a correct algorithm for **PRIME** terminates on any input q, returning YES if and only if q is a prime number; the output is YES for the instance above. An **incorrect** algorithm might not halt on some of the inputs, or it might halt but with a wrong output. Notice that a problem can admit many correct algorithms.

Pseudocode. We describe algorithms using a natural and intuitive notation, called **pseudocode**. It is close in spirit with Pascal and C notations, but we are not going to respect any strict syntax. A pseudocode algorithm can contain explanations and even some steps in plain English, or whatever convention or method that is clear, concise and expressive to specify a given algorithm. For example, if one wants to add all the elements in an array of natural numbers $A[1, ..., n]$, one can write the following pseudocode algorithms:

```
SUM(A)
S ← 0
for i ← 1 to length[A] do
    S ← S + A[i]
return S
```

or

```
SUM(A)
S ← A[1]
for i ← 2 to n (where n is the length of A)
    S ← S + A[i]
```

or even

```
SUM(A)
let S be the sum of all n elements in A
```

when calculating the sum of numbers in A is not the essential part of the algorithm and you do not want to bother with it. We try to be consistent in using a left arrow $\leftarrow$ for assignments and = for equality test; we take the liberty to use a series of left arrows for multiple assignments, such as $x \leftarrow y \leftarrow 1$. We generally use appropriate indentation to distinguish the instructions that are part of a loop. The pseudocode notation will become clearer shortly, reading the algorithms that follows. However, you should always have in mind that your pseudocode algorithm should be very easily translated into a real programming language.

During this course, we will look for algorithms that are *correct* and *efficient*. You will learn how to prove that an algorithm is correct and how to measure its efficiency. We will not put emphasis on formal correctness proofs because they may be very tedious and not worth the effort in practice.

1.1 Some Examples of Algorithms

In this section we present three simple algorithms, two for the problem **SORTING** and another for the problem **PRIME**.

1.1.1 Insertion Sort

Perhaps the simplest sorting algorithm is *insertion sort*, which starts with an empty sequence and then incrementally *inserts* the elements $a_1, a_2, ..., a_n$ such that the sequence stays sorted. Insertion sort works the way many card players sort their hand: start with an empty left hand and the cards face down on the table; then remove one card at a time from the table and insert it into the correct position in the left hand. For example, the sequence $\langle 17, 21, 8, 27, 3, 15, 21, 9, 7 \rangle$ is sorted in the following steps:

$$\langle \bullet\ 17, 21, 8, 27, 3, 15, 21, 9, 7 \rangle$$
$$\langle \underline{17,} \bullet\ 21, 8, 27, 3, 15, 21, 9, 7 \rangle$$
$$\langle \underline{17, 21,} \bullet\ 8, 27, 3, 15, 21, 9, 7 \rangle$$
$$\langle \underline{8, 17, 21,} \bullet\ 27, 3, 15, 21, 9, 7 \rangle$$
$$\langle \underline{8, 17, 21, 27,} \bullet\ 3, 15, 21, 9, 7 \rangle$$
$$\langle \underline{3, 8, 17, 21, 27,} \bullet\ 15, 21, 9, 7 \rangle$$
$$\langle \underline{3, 8, 15, 17, 21, 27,} \bullet\ 21, 9, 7 \rangle$$
$$\langle \underline{3, 8, 15, 17, 21, 21, 27,} \bullet\ 9, 7 \rangle$$
$$\langle \underline{3, 8, 9, 15, 17, 21, 21, 27,} \bullet\ 7 \rangle$$
$$\langle \underline{3, 7, 8, 9, 15, 17, 21, 21, 27} \rangle$$

The interested reader may consult `http://al.ei.tuat.ac.jp/~sekisita/isort-e.html` for a nice Java applet for insertion sort.

There may be many more or less equivalent ways to write the insertion sort algorithm in pseudocode. The following is perhaps the simplest:

INSERTION-SORT1(A)
for $i \leftarrow 1$ **to** $length[A] - 1$ **do**
  for $j \leftarrow i + 1$ **to** 2 **do**
    if $A[j] < A[j-1]$ **then** $swap(A[j], A[j-1])$

where $swap$ is the usual function that swaps the values of two locations. When the algorithm terminates, A will contain the sorted array.

The following is a slightly more efficient pseudocode for insertion sort:

INSERTION-SORT2(A)
for $i \leftarrow 2$ **to** $length[A]$ **do**
  $temp \leftarrow A[i]; \; j \leftarrow i - 1$
  while $j > 0$ and $A[j] > temp$ **do**
    $A[j+1] \leftarrow A[j]; \; j \leftarrow j - 1$
  $A[j+1] \leftarrow temp$

What does it mean for an algorithm to be better than another, or that an algorithm is efficient? We'll see in the next section how we can measure the efficiency of an algorithm, and thus compare various algorithms. This is of great practical importance because in almost all concrete situations, a certain problem admits more then one algorithm.

1.1.2 Merge Sort

Many algorithms are *recursive*, that is, in order to calculate their output they call themselves one or more times to first solve related subproblems usually for smaller inputs, and then combine the outputs of the subproblems to generate the output of the original problem. This approach is also called *divide-and-conquer* or *divide-et-impera*, and it consist of the following steps:

Divide the problem into a number of related subproblems;

Conquer the problems by recursively solving them, using the same strategy; there will be a moment when the subproblems become small enough to calculate their output without recursion;

Combine the outputs of the subproblems to generate the output of the original problem.

Merge sort is a sorting algorithm in this category, which can be described in a few words as follows: in order to sort $A[1, ..., n]$, first split the elements in two subsets with approximately the same number of elements[1] (divide) and sort them (conquer) in $A[1, ..., [\frac{n}{2}]]$ and $A[[\frac{n}{2}] + 1, ..., n]$, respectively, and then merge them (combine). To merge two sorted arrays, one may keep comparing the two smallest elements of the two arrays, outputting and removing the smaller one until one of the two arrays becomes empty, and then just output the remaining elements of the other array. We let it as an exercise for the reader (see Exercise 1.4) to write pseudocode for a procedure MERGE(A, i, j, k), where A is an array and $i \leq j < k$ are indices such that $A[i, ..., j]$ and $A[j+1, ..., k]$ are sorted, which returns all the $k - i + 1$ elements in a single sorted subarray that replaces the current subarray $A[i, ..., k]$. For example, if $A[1, ..., 9] = \langle 3, 8, 17, 21, 27, 7, 9, 15, 21 \rangle$, $i = 1$, $j = 5$, and $k = 9$, then MERGE(A, i, j, k) combines the subarrays $A[1, ..., 5] = \langle 3, 8, 17, 21, 27 \rangle$ and $A[6, ..., 9] = \langle 7, 9, 15, 21 \rangle$ as follows:

$$
\begin{array}{rcrl}
\langle \underline{3}, 8, 17, 21, 27 \rangle & , & \langle 7, 9, 15, 21 \rangle & \longrightarrow \langle \rangle \\
\langle 8, 17, 21, 27 \rangle & , & \langle \underline{7}, 9, 15, 21 \rangle & \longrightarrow \langle 3 \rangle \\
\langle \underline{8}, 17, 21, 27 \rangle & , & \langle 9, 15, 21 \rangle & \longrightarrow \langle 3, 7 \rangle \\
\langle 17, 21, 27 \rangle & , & \langle \underline{9}, 15, 21 \rangle & \longrightarrow \langle 3, 7, 8 \rangle \\
\langle 17, 21, 27 \rangle & , & \langle \underline{15}, 21 \rangle & \longrightarrow \langle 3, 7, 8, 9 \rangle \\
\langle \underline{17}, 21, 27 \rangle & , & \langle 21 \rangle & \longrightarrow \langle 3, 7, 8, 9, 15 \rangle \\
\langle 21, 27 \rangle & , & \langle \underline{21} \rangle & \longrightarrow \langle 3, 7, 8, 9, 15, 17 \rangle \\
\langle \underline{21}, 27 \rangle & , & \langle \rangle & \longrightarrow \langle 3, 7, 8, 9, 15, 17, 21 \rangle \\
\langle \underline{27} \rangle & , & \langle \rangle & \longrightarrow \langle 3, 7, 8, 9, 15, 17, 21, 21 \rangle \\
\langle \rangle & , & \langle \rangle & \longrightarrow \langle 3, 7, 8, 9, 15, 17, 21, 21, 27 \rangle
\end{array}
$$

[1] In n is odd then the first set has one more element.

Then the following is a correct pseudocode for merge sort:

MERGE-SORT(A, i, k)
if $i \geq k$ **then return**
$j \leftarrow [\frac{i+k}{2}]$
MERGE-SORT(A, i, j)
MERGE-SORT$(A, j+1, k)$
MERGE(A, i, j, k)

It is clear now that MERGE-SORT$(A, 1, n)$ correctly sorts the array $A[1, ..., n]$.

1.1.3 Prime

A natural number is prime if and only if it has no divisors except 1 and itself. Thus, the following is an immediate algorithm for testing if a natural number q is prime:

PRIME1(q)
for $i \leftarrow 2$ **to** $q - 1$ **do**
  if i $divides$ q **then return** NO
return YES

We assumed that the command **return** halts the algorithm and outputs the indicated value. We also used a function $divides$ which returns true if and only if its first argument divides the second one.

A brief analysis of the algorithm above indicates that it tests at least twice more indices i than needed. Indeed, a number q cannot have any divisor larger than $q/2$, so the algorithm can be immediately modified to be twice as efficient. Moreover, even $q/2$ is too large! The square root of q is a correct upper limit for potential divisors of q. Thus, the following algorithm is still correct and more efficient than the previous one:

PRIME2(q)
for $i \leftarrow 2$ **to** $\sqrt{q}$ **do**
  if i $divides$ q **then return** NO
return YES

Notice that PRIME2 is more than 100 times faster than PRIME1 when q is a prime number of five digits!

1.2 Analyzing Algorithms

We saw above that a simple analysis of PRIME1 yielded a more than 100 times more efficient algorithm. That was a very simple algorithm and our analysis was straightforward; unfortunately, the real algorithms are more complex and their analysis can be very complicated, especially if formal analysis is desired. Since this is an undergraduate course, our approach is to analyze the algorithms as intuitively as possible, still capturing the essential and interesting phenomena, avoiding the common pitfalls.

There may be many resources to be analyzed at an algorithm, such as running time, required memory, communication bandwidth, number of gates used, etc. However, in this course we are only interested in running time, that is, the expected execution time of an algorithm. It is of course desirable, and you'll get extra-points for that, to choose the algorithm which requires less memory or *space* when many algorithms having relatively the same running time are available. An example can be the computation of the n-th Fibonacci number, where the sequence of Fibonacci numbers $1, 1, 2, 3, 5, 8, 13, 21, 34, ...$, is recurrently defined as follows:

$$\begin{cases} f(0) = f(1) = 1, \\ f(n) = f(n-1) + f(n-1), \text{for all } n \geq 2. \end{cases}$$

An immediate efficient algorithm to compute the n-th Fibonacci number can be the following:

FIBONACCI1(n)
$f(0) \leftarrow f(1) \leftarrow 1$
for $i \leftarrow 2$ **to** n **do**
    $f(i) \leftarrow f(i-1) + f(i-2)$
return $f(n)$

which requires an array $f[0..n]$ of n natural numbers, so it requires a space of n locations[2]. The same problem can be solved by an algorithm requiring the same running time, but only 3 locations instead of n, as follows:

FIBONACCI2(n)
$x \leftarrow y \leftarrow 1$
for $i \leftarrow 2$ **to** n **do**
    $z \leftarrow y + x;\ x \leftarrow y;\ y \leftarrow z$
return y

or even by an algorithm requiring only 2 locations:

FIBONACCI3(n)
$x \leftarrow y \leftarrow 1$
for $i \leftarrow 2$ **to** n **do**
    $y \leftarrow y + x;\ x \leftarrow y - x$
return y

Nevertheless, we appreciate to see such solutions and are going to give extra-credit for them, but you should be aware that this is not the goal of this course. We'll concentrate on running time of algorithms rather than on memory efficiency. As we'll shortly see, all the three Fibonacci algorithms above have the same running time, which is linear.

1.2.1 RAM Model

Before we can analyze algorithms, we must have an abstract model of machine on which we can execute our algorithms. It is already well-accepted to assume a generic one-processor, *random-access machine (RAM)* model of computation, on which instructions are executed one after another, with no concurrent operations. In the RAM model, each simple operation ($+$, $-$, $*$, $/$, $=$, $\leftarrow$, $>$, $divides$, **if**, function calls) or memory access takes 1 time step; there is as much plain memory as needed and there is no cache memory. In general, it suffices to assume that a number needs exactly one location of memory to be represented in the abstract machine. However, there are situations where the representation of a number plays an important role, especially in problems whose input is a number, such as **PRIME**. In such situations, we assume that a number n needs $\log n$ space in RAM; this convention is based on the fact that a number is represented in a computer by a sequence of digits, whose length is $\log n$ when the number is represented in base 2. The loops are those which significantly increase the execution time of an algorithm, because in general they depend on the size of the input. The *computational time* (or the *execution time*, or the *running time*, or simply the *complexity*) of an algorithm is the number of time steps it takes to execute on the RAM model, as function of the size of the input.

1.2.2 Insertion Sort

In this subsection we analyze the algorithm INSERTION-SORT1. Before we proceed further, we make the assumption that all numbers to be sorted can be represented using an arbitrary but fixed number of memory units, say k. Then the size of the input $A = \langle a_1, a_2, ..., a_n \rangle$ of INSERTION-SORT1, denoted $|A|$, is $n \cdot k$.

Notice that the step 3 of the algorithm can take either 3 time steps or $6 + t_{swap}$ time steps, depending on whether the condition $A[j] < A[j-1]$ is false or true, where t_{swap} is the time taken by the function $swap$ to change the two

[2] In fact there is one more needed location, the one for the variable i.

elements. On the other hand, the step 3 of the algorithm is executed $(n-1)+(n-2)+\cdots+2+1$ times, or in a more compact notation, $\sum_{i=1}^{n-1} i$ times. We'll see in the next chapter that this sum is $n(n-1)/2$. Therefore, the total computational time of INSERTION-SORT1, written $T(\text{INSERTION-SORT1})$ verifies the property

$$3 \cdot \frac{n(n-1)}{2} \leq T(\text{INSERTION-SORT1}) \leq (6+t_{swap}) \cdot \frac{n(n-1)}{2}.$$

Since the execution time of an algorithm is a function of its input size and since $n = \frac{|A|}{k}$, we obtain that

$$3 \cdot \frac{\frac{|A|}{k} \cdot (\frac{|A|}{k} - 1)}{2} \leq T(\text{INSERTION-SORT1}) \leq (6+t_{swap}) \cdot \frac{\frac{|A|}{k} \cdot (\frac{|A|}{k} - 1)}{2},$$

that is,

$$\frac{3}{k^2} \cdot (|A|^2 - k \cdot |A|) \leq T(\text{INSERTION-SORT1}) \leq \frac{6+t_{swap}}{k^2} \cdot (|A|^2 - k \cdot |A|).$$

Thus, we have a lower and an upper limit for the expected total execution time. Nevertheless, the lower and upper limits look complicated and do not give us much intuition about the efficiency of the algorithm. For this reason, we'll introduce new notations in the next chapter, allowing us to give a more compact estimation of $T(\text{INSERTION-SORT1})$.

For now, let us consider a simplifying abstraction. Since we are interested in computational times of algorithms for large sizes of inputs, we'll pay special attention to the *order of growth* of the running time. Thus, we consider only the leading term $a_k n^k$ of a formula $a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$ since the lower order terms are relatively insignificant for large values of n. We'll also ignore the constant coefficient a_k of the leading term because it is also insignificant when n is very large. Then we say that the running time $T(\text{INSERTION-SORT1})$ of insertion sort is $\Theta(|A|^2)$, which is the same as $\Theta(n^2)$, where n is the number of elements to be sorted. We use the Θ-notation only informally in this chapter; it will be precisely defined in the next chapter.

A lesson one should learn from this example is that one may need some mathematical notions in order to analyze even simple algorithms. For example, one has to know how to calculate sums of the form $1+2+\cdots+(n-1)$; to analyze more complex algorithms, one needs to calculate even more complicated sums, such as $1^k+2^k+\cdots+(n-1)^k$ for any $k > 1$, $1\cdot 2 + 2\cdot 3 + \cdots + (n-1)\cdot n$, etc. We'll cover a few of these in the next chapter, at least those that seem necessary for the rest of this course.

1.2.3 Merge Sort

The analysis of divide and conquer algorithms in general follows a straightforward pattern, closely related to their three features, divide, conquer, and combine. The first step is to obtain an appropriate *recurrence* for the running time and the second step is to solve that recurrence. We'll only concentrate on the first step in this chapter, letting the second one for the next chapter.

Suppose that $\mathcal{A}$ is a divide and conquer algorithm which divides the problem into a subproblems and then combines their solutions to generate the solution of the original problem. If for an input of size n the a subproblems have sizes $n_1, n_2, ..., n_a$ and the time needed to divide the problem into the a subproblems and then to combine their solutions is $f(n)$, then

$$T(n) = T(n_1) + T(n_2) + \cdots + T(n_a) + f(n),$$

where $T(n)$ denotes the running time of the algorithm on an input of size n. It is often the case that the subproblems have the same size which is a fraction of n, say n/b, in which case the recurrence formula is

$$T(n) = aT(n/b) + f(n).$$

When the size of the subproblems become small enough, say smaller than a constant c, their output can be generated in a number of steps which doesn't depend on the size of the original problem; in this case we say that $T(n) = \Theta(1)$ for $n < c$. In the next chapter, we'll learn how to solve such recurrences, and hence how to calculate the running time of divide and conquer algorithms.

It can be easily seen that MERGE(A, i, j, k) runs in $\Theta(n)$, where n is the total number of elements to be merged, that is, $k - i + 1$. Applying the technique above to merge sort which is split in two subproblems of size $n/2$ whose solutions need $\Theta(n)$ time to be combined, we get the recurrence formula

$$T(n) = 2T(n/2) + \Theta(n).$$

We'll learn in the next chapter that the solution of this recurrence is $\Theta(n \log n)$, which is therefore the running time of merge sort.

1.2.4 Prime

It is an interesting experiment to compare the computational times of INSERTION-SORT1 (which we just proved to be $\Theta(n^2)$) and PRIME2. At a first sight, PRIME2 seems to be much more efficient because it consists of only one loop, from 1 to $\sqrt{n}$, so its computational time seems to be $\Theta(\sqrt{n})$, which is of course much faster than $\Theta(n^2)$. However, this reasoning is *wrong*! It is a common pitfall for beginners to fail in calculating the correct size of the input, thus failing in calculating the correct computational time of an algorithm, because it is a function of the size of the input.

According to our conventions for RAM, the size of the input q is $\log q$. Let n be $\log q$, that is, $q = 2^n$. Then the computational time of PRIME2 is

$$T(\text{PRIME2}) = \Theta(\sqrt{q}) = \Theta(\sqrt{2^n}) = \Theta(2^{\frac{n}{2}}).$$

We will see in the next section that $\Theta(2^{\frac{n}{2}})$ is larger than $\Theta(n^2)$, so PRIME2 is a slower algorithm than INSERTION-SORT1, in the sense that it takes longer to terminate on an input of the same size.

1.2.5 Best, Worst, and Average-Case Analysis

An algorithm can run in different number of steps for different inputs of the same size. We saw in Subsection 1.2.2 that INSERTION-SORT1 could have different computational times, depending on whether the condition $A[j] < A[j-1]$ was true or false. We gave lower and upper limits for the number of steps needed, and showed that the two limits were different only by an order of a constant, so we could still deduce that the running time of INSERTION-SORT1 is *always* $\Theta(n^2)$.

It is not always the case that our algorithms have the same running time (in Θ-notation) for any input. For example, if the input array A is already sorted then INSERTION-SORT2 runs in time $\Theta(n)$ because the condition of the **while** statement in step 3 is never true, so the steps 2,3,4 always need together $\Theta(1)$ time.

There can be defined at least three functions on the input size of an algorithm $\mathcal{A}$:

Worst-Case Complexity is the function of n giving the *maximum* number of steps needed by $\mathcal{A}$ to terminate on any input of size n;

Best-Case Complexity is the function of n giving the *minimum* number of steps needed by $\mathcal{A}$ to terminate on any input of size n;

Average-Case Complexity is the function of n giving the *average* number of steps needed by $\mathcal{A}$ to terminate on any input of size n.

The average-case complexity of an algorithm is usually harder to calculate than the other two functions, because it involves *all* possible executions of the algorithm. The most usual one is worst-case complexity and this is the one we are concentrating on in this course. When unless specified, by analyzing an algorithm we mean calculating its worst-case complexity.

Preparatory Exercises

Exercise 1.1 Give an informal correctness proof for INSERTION-SORT2.

Exercise 1.2 Why is INSERTION-SORT2 "slightly more efficient" then INSERTION-SORT1 in Subsection 1.1.1?

Exercise 1.3 Simulate the execution of merge sort on the input $A[1,...,9] = \langle 17, 21, 8, 3, 27, 15, 21, 9, 7\rangle$ as we did in Subsection 1.1.1 for insertion sort.

Exercise 1.4 Write and analyze pseudocode for MERGE(A, i, j, k) in Subsection 1.1.2.

Proof: There may be many pseudocode algorithms for the same problem, but we expect all of them to use auxiliary memory. Notice that our RAM computation model allows us to use as much memory as we need. Then the following is a possible pseudocode:

MERGE(A, i, j, k)
$i' \leftarrow i;\ j' \leftarrow j+1;\ counter \leftarrow 0$
while $i' \leq j$ or $j' \leq k$ **do**
  if $(i' > j)$ or $(j' \leq k$ and $A[i'] \geq A[j'])$ **then**
    $counter \leftarrow counter + 1;\ A'[counter] = A[j'];\ j' \leftarrow j' + 1$
  if $(j' > k)$ or $(i' \leq j$ and $A[i'] \leq A[j'])$ **then**
    $counter \leftarrow counter + 1;\ A'[counter] = A[i'];\ i' \leftarrow i' + 1$
for $i' \leftarrow 1$ **to** $counter$ **do**
  $A[i + i' - 1] \leftarrow A'[i]$

where $A'[1,...,k-i+1]$ is an auxiliarry array.

Therefore, each iteration of the while loop increments $counter$ and either i' or j' by one. Since there are $k-i+1$ elements in total to be merged and since the while loop is executed until both $i' \geq j$ and $j' \geq k$, we deduce that the while loop is executed exactly $k-i+1$ times and that this is the value of $counter$. If we let n denote the total number of elements in the input, that is, $n = k - i + 1$, then the running time of MERGE is $\Theta(n)$.

Exercise 1.5 Explain why $\sqrt{q}$ is a correct upper bound for potential divisors of q in PRIME2 in Subsection 1.1.3.

Proof: It suffices to show that if q has divisors different than 1 and q then it has at least one divisor between 2 and $\sqrt{q}$. We prove it by contradiction: suppose that $q = x \cdot y$ and that $x > \sqrt{q}$ and $y > \sqrt{q}$; then $x \cdot y > \sqrt{q} \cdot \sqrt{q} = q$, that is, $q > q$, contradiction.

Exercise* 1.6 Why did we assume that numbers can be represented on an arbitrary but *fixed* number of memory locations in the analysis of insertion sort in Subsection 1.2.2? Calculate the running time of INSERTION-SORT1 on an abstract machine where the comparison operation $>$ takes a number of time steps proportional with the size of the compared numbers, when the size of the n elements in n^k for some $k \geq 0$. (Hint: If the numbers can have any size, in particular if their size depends on n, then the RAM model presented in Subsection 1.2.1 is not appropriate anymore because the comparison operation $>$ cannot be executed in 1 time step, but in a number of time steps proportional with the size of the representation of the numbers.)