1.1.10(a)
Many possibilities. See:
http://en.wikipedia.org/wiki/Euclidean_algorithm and
http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm

Hw 1 -key

1.2.3
A. (as long as we know how to compute square root of any positive integer)

1.2.9
Fix the loop counters such that each pair is considered only once, and pre-compute the difference between two elements:

```
Algorithm: MyMinDist(A[0...n-1])
//Input: Array A[0...n-1] of numbers
//Output: Smallest distance d between two of its elements
dmin<-inf
for i <- 0 to n-2 do
  for j <- i+1 to n-1 do
    temp <- |A[i] - A[j]|
    if temp < dmin
      dmin <- temp
return dmin
```

other solutions might include presorting and then doing a linear scan though the array.

```
// pre-sort (see Levitin: page 126)
Algorithm Mergesort(A[0...n-1])

Algorithm MyMinDist(A[0...n-1])
//Input: Sorted array A[0...n-1] of numbers
//Output: Smallest distance d between two of its elements
dmin <- inf
for i <- 0 to n-2 do
    temp <- |A[i] - A[i+1]|
    if temp < dmin
      dmin <- temp
return dmin
```

1.3.10
extra credit.

1.4.9 or 1.4.10
1.4.9
a. a priority queue
b. a queue
c. a stack with reverse Polish notation

1.4.10
easy one:
search for each successive letter of the first work in the second one. If the search is successful, delete the first occurrence of the latter in the second word, and stop otherwise.

better:
sort the letters of each word and then compare them in a single parallel scan.

other:
Create letter vectors for each work; scan each work and for each letter occurrence, add one to letter in appropriate letter vector (representing letter counts). Compare two vectors.

many more exist...

2.1.6
Before applying a sorting algorithm, compare the adjacent elements of its input: if a_i =< a_i+1 for every i=0...n-2 stop. Generally, it is not a worthwhile addition because it slows down the algorithm on all but very special inputs. Note some sorting algorithms (Bubble and insertion) intrinsically incorporate this test in the body of the algorithm.

2.1.7
a. T(2n)/T(n) = [c_m 1/3 (2n)^3]/ [c_m 1/3 n^3] = 8 (cm is the time of one multiplication.
b.We can estimate the running time for solving systems of order n on the old computer and that of order N on the new computer as T_old(n)=c_m 1/3 n^3 and T_new(N)=10^-3 c_m 1/3 N^3, respectively, where c_m is the time of one multiplication on the old computer. Replacing T_old(n) and T_new(N) by these estimates in the equation T_old(n) = T_new(N) yields c_m 1/3 n^3 = 10^-3 c_m 1/3 N^3 or N/n = 10.

2.2.5
5 lg(n+100)^10, ln^2(n), n^(1/3), 0.001n^4 + 3n^3 + 1, 3^n, 2^(2n), (n-2)!

2.2.9
a. Theta(n log(n)) + O(n) = Theta(n log(n))
b. Theta(n)

2.3.5(a-d)
a. An element comparison

b. C(n) = Sum_(i=1)^(n-1) (2) = 2(n-1)

c. Theta(n)

d. Obvious improvements for some inputs it so replace the two if-statements by the following:
```
if A[i]< minval minval <- A[i]
else if A[i]> maxval maxval <- A[i]
```

Another improvement, both more subtle and substantial, is based on the observation that it is more efficient to update the minimum and maximum values seen so far not for each element but for a pair of two consecutive elements. If two such elements are compared with each other first, the updates will require only two more comparisons for the total of three comparisons per pair. note that the same improvement can be obtained by a divide-and-conquer algorithm.