Proposed solution to problem 1

(a) The answers:

(1) $f = O(g)$: if $\alpha > 1$.
(2) $f = \Omega(g)$: if $\alpha \leq 1$.
(3) $g = O(f)$: if $\alpha \leq 1$.
(4) $g = \Omega(f)$: if $\alpha > 1$.

(b) $T(n) = \Theta(2^{\frac{n}{2}}) = \Theta(\sqrt{2}^n)$

(c) $T(n) = \Theta(n \log n)$

(d) $\Theta(n \log n)$

(e) $\Theta(n \log n)$

Proposed solution to problem 2

(a) A call *mystery*($v,\ 0,\ v.size()-1,\ m$) permutes the elements of v so that (at least) the m first positions of v contain the m smallest elements, sorted in increasing order.

(b) When one makes a call *mystery*($v,\ 0,\ n-1, n$), first **int** q = *partition* ($v,\ l,\ r$) is executed with $l = 0$ and $r = n-1$, with cost $\Theta(n)$. Moreover, as the vector contains different integers sorted in increasing order and the function *partition* takes as a pivot the leftmost element, we have $q = 0$. Hence we have that the call *mystery*($v,\ l,\ q,\ m$) is made with $q = l = 0$ and so takes constant time. Besides, $p = 1$. If $n > 1$ finally a recursive call *mystery*($v,\ q+1,\ r,\ m-p$) is made, where $q+1 = 1$, $r = n-1$ and $m-p = n-1$.

In turn, when executing this call the cost of **int** q = *partition* ($v,\ l,\ r$) is $\Theta(n-1)$, again the cost of *mystery*($v,\ l,\ q,\ m$) is $\Theta(1)$, and if $n-1 > 1$ again a recursive call *mystery*($v,\ q+1,\ r,\ m-p$), is made, where $q+1 = 2$, $r = n-1$ and $m-p = n-2$.

Repeating the argument, we see that the accumulated cost is

$$\Theta(n) + \Theta(n-1) + \ldots + \Theta(1) = \Theta(\sum_{i=1}^{n} i) = \Theta(n^2).$$

Proposed solution to problem 3

(a) A possible solution:

```
vector<bool> prod(const vector<bool>& x, const vector<bool>& y) {

  if (x.size() == 0 or y.size() == 0) return vector<bool>();

  vector<bool> z = twice(twice(prod( half(x), half(y) )));
  vector<bool> one = vector<bool>(1, 1);

  if      (x.back() == 0 and y.back() == 0) return z;
  else if (x.back() == 1 and y.back() == 0) return sum(z, y);
  else if (y.back() == 1 and x.back() == 0) return sum(z, x);
  else {
    vector<bool> x2 = twice(half(x));
    vector<bool> y2 = twice(half(y));
    return sum(sum(sum(z, x2), y2), one);
  }
}
```

Let $T(n)$ be the cost of *prod*(x,y) if $n = x.size() = y.size()$. Only one recursive call is made, over vectors of size $n-1$. The non-recursive work has cost $\Theta(n)$. So the cost follows the recurrence

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

According to the Master Theorem of Subtractive Recurrences, the solution of this recurrence behaves asymptotically as $\Theta(n^2)$. Hence, the cost is $\Theta(n^2)$.

(b) Karatsuba algorithm, which has cost $\Theta(n^{\log 3})$.

Proposed solution to problem 4

(a) A possible way of completing the code:

```
bool search1 (int x, const vector<vector<int>>& A, int i, int j, int n) {
  if (n == 1) return A[i][j] == x;
  int mi = i + n/2 − 1;
  int mj = j + n/2 − 1;
  if (A[mi][mj] < x) return
      search1 (x, A, mi+1,    j, n/2) or
      search1 (x, A, mi+1, mj+1, n/2) or
      search1 (x, A,    i, mj+1, n/2);
  if (A[mi][mj] > x) return
      search1 (x, A, mi+1,    j, n/2) or
      search1 (x, A,    i,    j, n/2) or
      search1 (x, A,    i, mj+1, n/2);
  return true;
}
```

We note that the result of calling *search1* $(x,\ A,\ 0,\ 0,\ N)$ is **true** if and only if x occurs in A.

To analyse the cost of this call, first of all we study the general case of calling *search1* $(x,\ A,\ i\ ,\ j\ ,\ n)$ as a function of n. Let $T(n)$ be the cost in the worst case of this call. As in the worst case 3 calls are made with last parameter $n/2$ and the cost of the non-recursive work is constant, we have the recurrence:

$$T(n) = 3T(n/2) + \Theta(1)$$

Applying the Master Theorem of Divisive Recurrences, we have $T(n) = \Theta(n^{\log_2 3})$.

Hence, the cost of calling *search1* $(x,\ A,\ 0,\ 0,\ N)$ is $\Theta(N^{\log_2 3})$.

(b) It is correct. Let us see that the loop maintains the following invariant: if x occurs in A, then the occurrence is between rows 0 and i (included), and between columns j and $N-1$ (included). At the beginning of the loop, the invariant holds. And at each iteration it is preserved:

- If $A[i][j] > x$ then x cannot occur at row i: by the invariant we have that if x occurs then it must be in a column from j to $N-1$. But if $j \leq k < N$ then $A[i][k] \geq A[i][j] > x$.
- If $A[i][j] < x$ then x cannot occur at column j: by the invariant we have that if x occurs then it must be in a row from 0 to i. But if $0 \leq k \leq i$ then $A[k][j] \leq A[i][j] < x$.

Finally, if the program returns **true** with the **return** of inside the loop, the answer is correct as $A[i][j] = x$. If it returns **false** with the **return** of outside the loop, then $i < 0$ or $j \geq N$. In any case, by the invariant we deduce that x does not appear in A.

(c) At each iteration either i is decremented by 1, or j is incremented by 1, or we return. Moreover, initially i has value $N-1$, and at least has value 0 before exiting the loop. Similarly, at the beginning j has value 0 and at most has value $N-1$ before exiting the loop. As in the worst case we can make $2N-1$ iterations and each of them has cost $\Theta(1)$, the cost is $\Theta(N)$.