Proposed solution to problem 1

(a) Insertion sort has cost $\Theta(n)$ when the vector is already sorted, and cost $\Theta(n^2)$ when it is sorted backwards. So the average cost is:

$$(1 - \frac{\log n}{n})\Theta(n) + \frac{\log n}{n}\Theta(n^2) = \Theta(n - \log n + n \log n) = \Theta(n \log n)$$

(b) The function returns whether n is a prime number. The cost is determined by the loop, which makes $O(\sqrt{n})$ iterations, each of which has constant cost. Thus the cost is $O(\sqrt{n})$.

(c) Each of the numbers that are multiplied in $n!$ are less than or equal to n. Since we multiply n of them, we have $n! \leq n^n$ for all $n \geq 1$. So $n! = O(n^n)$, taking $n_0 = 1$ and $C = 1$.

(d) Except for 1, all numbers that are multiplied in $n!$ are greater than or equal to 2. Since we multiply $n - 1$ of them, we have $n! \geq 2^{n-1} = \frac{1}{2} \cdot 2^n$ for all $n \geq 1$. So $n! = \Omega(2^n)$, taking $n_0 = 1$ and $C = \frac{1}{2}$.

Proposed solution to problem 2

(a) A possible solution:

```
int top_rec (const vector<int>& a, int l, int r) {
  if (l + 1 ≥ r) {
    if (a[l] < a[r]) return r;
    else             return l;
  }
  int m = (l+r)/2;
  if (a[m−1] > a[m]) return top_rec(a, l, m−1);
  if (a[m+1] > a[m]) return top_rec (a, m+1, r);
  return m;
}

int top(const vector<int>& a) {
  return top_rec (a, 0, a.size()−1);
}
```

Let $C(n)$ be the cost in time in the worst case of function *top_rec* on a vector of size n. In the worst case (for example, when the top is in one of the ends of the vector) a recursive call will be made on a subvector of size $n/2$, in addition to operations of constant cost (arithmetic computations, comparisons and assignments of integers, vector accesses). So we have the recurrence $C(n) = C(n/2) + \Theta(1)$, which by the master theorem of divisive recurrences has solution $C(n) = \Theta(\log n)$.

(b) A possible solution that uses the function **int** *top* (**const** *vector*<**int**>& *a*) of the previous exercise and the function *binary_search* of STL:

```
bool search(const vector<int>& a, int x) {
  int t = top(a);
  int n = a.size();
  if (binary_search(a.begin(), a.begin() + t,      x)) return true;
  if (binary_search(a.rbegin(), a.rbegin() + n - t, x)) return true;
  return false;
}
```

In another possible solution, the previous *search* function can be replaced by:

```
bool bin_search_inc(const vector<int>& a, int l, int r, int x) {
  if (l > r) return false;
  int m = (l+r)/2;
  if (a[m] < x) return bin_search_inc(a, m+1, r,   x);
  if (a[m] > x) return bin_search_inc(a,   l, m-1, x);
  return true;
}

bool bin_search_dec(const vector<int>& a, int l, int r, int x) {
  if (l > r) return false;
  int m = (l+r)/2;
  if (a[m] < x) return bin_search_dec(a,   l, m-1, x);
  if (a[m] > x) return bin_search_dec(a, m+1, r,   x);
  return true;
}

bool search(const vector<int>& a, int x) {
  int t = top(a);
  int n = a.size();
  if (bin_search_inc(a, 0, t-1, x)) return true;
  if (bin_search_dec(a, t, n-1, x)) return true;
  return false;
}
```

When function *search* searches in a vector of size n, in addition to calling function *top*, one or two binary searches are made, each of which has cost $O(\log n)$, and also operations of constant cost. So the cost of *search* is $O(\log n) + O(\log n) + O(1) = O(\log n)$. Moreover, if for example the top of the sequence is in one of the ends of the vector, then the cost is $\Theta(\log n) + O(\log n) + O(1) = \Theta(\log n)$. So the cost in the worst case is $\Theta(\log n)$.

Proposed solution to problem 3

(a) After m calls to function *reserve* on an initially empty vector, the capacity of the vector is of $C(m) = Am$ elements. So the number of calls to *reserve* after n calls to *push_back* is the least m such that $Am \geq n$, that is, $\lceil \frac{n}{A} \rceil$. Since A is constant, m is $\Theta(n)$.

(b) By induction.

- **Base case:** $m = 0$. We have that $\frac{BA^m - B}{A-1}|_{m=0} = \frac{B-B}{A-1} = 0 = C(0)$.
- **Inductive case:** assuming that it is true for m, let us show it is also true for $m+1$. By induction hypothesis, $C(m) = \frac{BA^m - B}{A-1}$. Then:

$$C(m+1) = AC(m) + B = A\frac{BA^m - B}{A-1} + B = \frac{BA^{m+1} - AB + AB - B}{A-1} = \frac{BA^{m+1} - B}{A-1}$$

(c) After m calls to function *reserve* on an initially empty vector, the capacity of the vector is of $C(m) = \frac{BA^m - B}{A-1}$ elements. So the number of calls to *reserve* after n calls to *push_back* is the least m such that $\frac{BA^m - B}{A-1} \geq n$, that is, $\lceil \log_A(\frac{n(A-1)+B}{B}) \rceil$. Since A and B are constants, m is $\Theta(\log n)$.

Proposed solution to problem 4

(a) A possible solution:

```
int stable_partition (int x, vector<int>& a) {
  int n = a.size ();
  vector<int> w(n);
  int i = 0;
  for (int y : a)
    if (y ≤ x) {
      w[i] = y;
      ++i;
    }
  int r = i−1;
  for (int y : a)
    if (y > x) {
      w[i] = y;
      ++i;
    }
  for (int k = 0; k < n; ++k)
    a[k] = w[k];

  return r;
}
```

The cost in time is $\Theta(n)$, since the vector is traversed 3 times, and each iteration of these traversals only requires constant time. The cost in space of the auxiliary memory is dominated by vector w, which has size n. Hence the cost in space is $\Theta(n)$.

(b) The function transposes the two subvectors of a from l to p and from $p+1$ to r: if before the call $a[l..r]$ is $A_l, \ldots, A_p, A_{p+1}, \ldots, A_r$, then after the call $a[l..r]$ is $A_{p+1}, \ldots, A_r, A_l, \ldots, A_p$.

(c) Solution:

```
int stable_partition (int x, vector<int>& a) {
  return stable_partition_rec (x, a, 0, a.size()−1);
}

int stable_partition_rec (int x, vector<int>& a, int l, int r) {
  if (l == r) {
    if (a[l] ≤ x) return l;
    else          return l−1;
  }
  int m = (l+r)/2;
  int p = stable_partition_rec (x, a, l, m);
  int q = stable_partition_rec (x, a, m+1, r);
  mystery(a, p+1, m, q);
  return p+q−m;
}
```

Let $C(n)$ be the cost of function *stable_partition_rec* on a vector of size $n = r - l + 1$ in the worst case. On the one hand two recursive calls are made on vectors of size $n/2$. On the other, the cost of the non-recursive work is dominated by function *mystery*, which takes time which is linear in the size of the vector that is being transposed. Using the hypothesis of the statement (which happens, for example, in a vector in which elements smaller and bigger than x alternate successively), this vector has size $\Theta(n)$. So we have the recurrence $C(n) = 2C(n/2) + \Theta(n)$, which by the master theorem of divisive recurrences has solution $\Theta(n \log n)$. In conclusion, the cost of function *stable_partition* on a vector of size n in the worst case is $\Theta(n \log n)$.