Proposed solution to problem 1

(a) $\Theta(\sqrt{n}\log n)$

(b) We compute:

$$\lim_{n\to\infty}\frac{\log(\log n^2)}{\log n}=\lim_{n\to\infty}\frac{\log(2\log n)}{\log n}=\lim_{n\to\infty}\frac{\log 2+\log(\log n)}{\log n}=$$

$$=\lim_{n\to\infty}\frac{\log 2}{\log n}+\lim_{n\to\infty}\frac{\log(\log n)}{\log n}=\lim_{n\to\infty}\frac{\log(\log n)}{\log n}$$

With the variable renaming $n=2^m$ we have that the previous limit is equal to

$$\lim_{m\to\infty}\frac{\log(\log 2^m)}{\log 2^m}=\lim_{m\to\infty}\frac{\log m}{m}=0$$

Hence it only holds that $\log(n)\in\Omega(\log(\log(n^2)))$

Proposed solution to problem 2

(a) It returns $f\circ g$, the function composition of f with g. The cost of *mystery* is the cost of the auxiliary function *mystery_aux*, that is given by the recurrence $T(n)=T(n-1)+\Theta(1)$, which has asymptotic solution $T(n)\in\Theta(n)$.

(b) It returns f^k. That is, a function such that $f^k(x)=\underbrace{f(f(\ldots(f}_{k}(x))))$. As a function of k, its cost is given by the recurrence $T(k)=T(k-1)+\Theta(1)$, with solution $\Theta(k)$.

(c)

```
vector<int> mystery_2_quick(const vector<int>& f, int k) {
  if (k == 0) {
    vector<int> r(f.size());
    for (int i = 0; i < f.size(); ++i) r[i] = i;
    return r;
  }
  else if (k%2 == 0) {
    vector<int> aux = mystery_2_quick(f,k/2);
    return mystery(aux,aux);
  }
  else {
    vector<int> aux = mystery_2_quick(f,k/2);
    return mystery(f,mystery(aux,aux));
  }
}
```

The recurrence that describes the cost of this function is $T(k)=T(k/2)+\Theta(1)$, which has solution $\Theta(\log k)$.

Proposed solution to problem 3

(a) It is not difficult to see that function *max_sum* essentially implements selection-sort, that we know has worst-case cost of $\Theta(m^2)$. The only difference is in the line where we update *sum*, that takes constant time and is executed m times. Hence, the total cost is $\Theta(m^2) + \Theta(m) = \Theta(m^2)$.

(b) If we understand the code, we can realize that it sorts the elements in S from larger to smaller and then multiplies them pairwise following this order. In order to improve the efficiency, we only have to sort the vector with *merge sort*, so that the cost is $\Theta(m \log m)$, and multiply the elements pairwise in the resulting order. The cost will be $\Theta(m \log m)$.

(c) Assume that x_0 i x_1 are the two largest elements in S and consider an expression that contains the products $x_0 * y$ and $x_1 * z$, for some $y, z \in S$. What we will do is to replace these two products by $x_0 * x_1$ and $y * z$. We know observe that $(x_0 * x_1 + y * z) - (x_0 * y + x_1 * z) = x_0(x_1 - y) + (y - x_1)z = x_0(x_1 - y) - (x_1 - y)z = (x_0 - z)(x_1 - y) > 0$. The last step is due to the fact that $x_0 > z$ and $x_1 > y$ since x_0 and x_1 are the largest elements in S, and they are all different. Hence the original expression was not maximum since the resulting expression after the replacement is larger.

Let us now prove the correctness of *max_sum* by induction on m:

- *Base case* ($m = 0$). The algorithm is correct since it returns 0, the maximum possible sum of products.
- *Induction step.* Let $m > 0$ and let us assume the induction hypothesis: the maximum expression for a set with $< m$ elements can be obtained by sorting the elements from larger to smaller and pairing them consecutively in this order. If we sort the m elements $x_0 > x_1 > x_2 > x_3 > \cdots > x_{m-1}$, we know, by the previous property, that the maximum expression contains the product $x_0 * x_1$ followed by an expression formed by the numbers $\{x_2, x_3, \ldots, x_{m-1}\}$. Obviously, this expression will be the largest we can form with $\{x_2, x_3, \ldots, x_{m-1}\}$ and by applying the induction hypothesis we know it will be $x_2 * x_3 + \cdots + x_{m-2} * x_{m-1}$. Hence, the maximum expression is $x_0 * x_1 + x_2 * x_3 + \cdots + x_{m-2} * x_{m-1}$, as we wanted to prove.

Proposed solution to problem 4

(a)

```
int f(const vector<int>& p, int l, int r){
  if (l + 1 ≥ r) return (p[l] ≤ p[r] ? l : r);
  else {
    int m = (l+r)/2;
    if (p[m] > p[m+1]) return f(p,m+1,r);
    else if (p[m−1] < p[m]) return f(p,l,m−1);
    else return m;
```

```
  }
}

pair<int,int> max_profit (const vector<int>& p) {
  return {f(p,0,p.size()-1), p.size()-1};
}
```

The cost of max_profit will be the cost of f, whose cost is given by the recurrence $T(n) = T(n/2) + \Theta(1)$, from which we obtain the cost $\Theta(\log n)$.

(b)

```
int max_profit (const vector<int>& p, int k) {
  int m = p[k];
  for (int i = k - 1; i >= 0; --i)
    m = min(m,p[i]);

  int M = p[k];
  for (int i = k + 1; i < p.size (); ++i)
    M = max(M,p[i]);

  return M - m;
}
```

(c) We can use a divide-and-conquer approach. Given a vector p we consider its middle point m and we split the vector into two equal-sized parts. Recursively, we compute the maximum profit if we buy and sell in the left part of the vector, and then, also recursively, the maximum profit if we buy and sell in the right part of the vector. Finally, using the function in b) we compute the maximum profit of a period that includes the middle point m (that is, we buy in the left part of the vector and we sell in the right). The final result is the maximum of the three computed profits.

We have designed a divide-and-conquer algorithm where there are two recursive calls where we half the size, and we then perform a linear amount of work in order to compute the maxim profit that includes point m. Hence, the recurrence that determines the cost is $T(n) = 2T(n/2) + \Theta(n)$, which has asymptotic solution $\Theta(n \log n)$.

Remark: there are more efficient solutions not necessarily based on divide and conquer.