Q1:

a)

First two lines each takes time $\Theta(1)$.

Inside the loop, the second line takes

$$\sum_{i=1}^{n} \Theta(\log i) = \theta(\log n!) = \theta(n \log n)$$

The first line also calculates MulWithInt, which also takes $\theta(n \log n)$ time. Since M grows by a factor of 10, S grows by adding M, the result of MulWithInt always have 1 more digit than S, so the first line takes

$$\begin{aligned} &\sum_{i=1}^{n} \Theta(\log i) + \sum_{i=1}^{n} \max\{\Theta(\log i - 1), \quad \Theta(\log(i))\} \\ &= \theta(n \log n) + \theta(\log n!) \\ &= \theta(n \log n) \end{aligned}$$

In total the algorithm takes time

$$2 \times \theta(1) + \theta(n \log n) + \theta(n \log n) = \theta(n \log n)$$

b)

```
PARSE(S[0 ... n-1]){
      if ((n-1) == 0) {
            return LONGINT(S[0]);
      }
      A = PARSE(S[0 ... (n-1)/2]);
      B = PARSE(S[(n-1)/2+1 ... n-1]);
      return ADD(A, MULWITH(B,10^((n-1)/2)));
}
```

Running time:

$2T\left(\frac{n}{2}\right)$ for assigning A,B; $T\left(\frac{n}{2}\right) + \max\left(\theta\left(\log\frac{n-1}{2}\right), \theta(\log(n-1))\right)$ for calculating ADD; $T(n)$ for the extra steps.

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \max\left(\theta\left(\log\frac{n-1}{2}\right), \theta(\log(n-1))\right) + T(n) \\ &= 3T\left(\frac{n}{2}\right) + \mathrm{T}(n) \\ &= \theta\left(n^{\log_2 3}\right) \end{aligned}$$

Q2:

The diameter is max{ the diameter of left tree,

the diameter of right tree,

2 + the longest path of the left tree + the longest path of the left tree}

This question can be solved by use a helper function which recursively returns a structure containing "length of the longest path" and "the diameter", then we can record the longest path and diameter in each node, and do the comparison.

```
structure pair{
//path for longest path, di for diameter
       int path=0;
       int di=0;
}

pair help(Tree T, Node v){
       pair child_l;
       pair child_r;
       if (LEFTCHILD(T,v) == null) {
              child_l = (0,0);
       } else {
              child_l = help(T,LEFTCHILD(T,v));
       }
       if (RightCHILD(T,v) == null) {
              child_r = (0,0);
       } else {
              child_r = help(T,RIGHTCHILD(T,v));
       }
       return pair(max(child_r.path, child_r.path) + 1,
                    max(child_l.di, child_r.di, 2 + child_l.path + child_r.path));
}

int diameter(T){
       return help(T, ROOT(T)).di;
}
```

There are three patterns of "diameter": it is the diameter of the left child, it is the diameter of the right child, or the diameter is the length from left deepest leaf to the right deepest leaf. The first two values will be stored in the "pair.di" field, and the last value can be computed by adding "pair.path" fields and 2. The diameter is the maximum of the three.

Function "help" runs once on every node of tree T, each run takes $\theta(1)$ time, the total run time is $\theta(n)$ where n is the number of nodes in tree T. $T(n) = 2T\left(\frac{n}{2}\right) + \theta(1) = \theta(n)$.

Q3:

a)

It is true.

Assume we want to make changes $c < d_{n-1}$. Since only coins we can use are d_n, we can get the optimal solutions by the greedy algorithm.

Assume we want to make changes $d_{n-1} \leq c < d_{n-2}$. Now we can use coins d_{n-1} and d_n. Let $d_{n-1} = kd_n$. For k $d_n$coins, we can replace them with 1 $d_n$coin and reduce the total number of coins, until the number of d_n coins are less than k. This is the solution that greedy algorithm will give us: take maximum number of d_{n-1} coins, then take d_n coins for the rest. This can be applied to all c that are between any consecutive denominations, so the greedy algorithm gives the optimal solutions for all $c < d_1$.

$c = d_1$ is a trivial case. The greedy algorithm will pick 1 d_1 coin as it is largest.

Assume we want to make changes $c > d_1$. Let $c = ad_1 + b$, where $b < d_1$, by division theorem a and b are unique. We can use a d_1 coins and find out changes for b using greedy algorithm. If we use less than a coins, for each 1 d_1 coin we gaved up, we need to use more than 1 d_2 coin to make that up since d_2 divides d_1, or even more coins with less denomination.

Hence the statement is true, the GreedyChange algorithm always returns an optimal solution to coin changing problem.

b)

false.

Suppose we have coins 9,4,1, and we want to make changes for 12.

Greedy solution will give 9+1+1+1 =12 with 4 coins, but an optimal solution will be 4+4+4=12 with 3 coins.

Q4:

Since the penalty is fixed, to minimize the penalty, we need to minimize the number of lates. The strategy is that try to finish the task that can be done before the deadline. If it cant be finished, don't start doing it and leave time for those which can be finished in time. After every deadline has passed, make up the tasks that are left.

```
int[] DEADLINE(int[] d, int n){

	//the sort step takes nlogn time, but can be omitted depending on the input d.
	//sorts d in increasing order.
	sort(d);

	int index_d = 0;
	int index_a = 0;
	ind index_a_back = n - 1;
	int[n] answer;
	for (int t = 0; (t < n && index_d < n && index_a < index_a_back); ++t){
		//task cannot be completed in time
		if (d[index_d]< t+1){
			answer[index_a_back] = d[index_d]
			++index_d;
			--index_a_back;
			--t;
		//task can be completed in time
		} else {
			answer[index_a] =d[index_d];
			++index_a;
			++index_d;
		}
	}
	return answer;
}
```

Starting from a certain time, we check the next the deadline by viewing the deadline array. There are only two possible answers: it can be finished in time or it cannot. If it can, we put them in schedule and advance in time; otherwise we check the next line, and put the current task at the back since it cant be finished anyway without penalty. All tasks will eventually be put in schedule, minimizing the number of tasks which will be late.

Sorting deadlines takes $\theta(n \log n)$ time, the loop runs over the deadlines, each time "index_d" is incremented by 1, and loop stops when "index_d" reaches n or sooner, so loop takes $\theta(n)$ time, the algorithm takes $\theta(n \log n)$ time if deadlines are not sorted, $\theta(n)$ time if deadlines are sorted.

Q5:

We divide both polynomials into half, recursively calculate the products of 4 smaller polynomials and sum them up to get original product.

```
// +,- operators have been overloaded.
// half_first() returns half of polynomial which has lower degree.
// half_first() returns rest half of polynomial, and lower its degree by half.
// polynomial.coeffs[i] access its ith coefficient.
// degree() returns the degree of a polynomial, returns -1 if polynomial = 0.
// shift(poly, i) multiplies polynomial poly by x^i.
Polynomial Polynomial::operator*(const Polynomial &rhs) const{
	if (degree() == 0){
		Polynomial temp(*this);
		temp.coeffs[0] *= rhs.coeffs[0];
		return temp;
	}
	Polynomial half1 = half_first();
	Polynomial half2 = half_last();
	Polynomial half1r = rhs.half_first();
	Polynomial half2r = rhs.half_last();
	Polynomial sum = (half1 + half2) * (half1r + half2r);
	Polynomial sum1 = half1 * half1r;
	Polynomial sum2 = half2 * half2r;
	sum = sum - sum1 - sum2;
	shift(sum2, 2*ceil((degree()+1)/2));
	shift(sum, ceil((degree()+1)/2));
	return sum + sum1 + sum2;
}
```

Assume all polynomials have degree 2d-1, further assume first polynomial $p_1 = ax^d + b$, second polynomial $p_2 = cx^d + e$, where a, b, c, e are polynomials with degree $d - 1$.

Then we need to compute

$$\begin{aligned} p_1 \times p_2 &= \big(ax^d + b\big)\big(cx^d + e\big) \\ &= acx^{2d} + (ae + bc)x^d + be \\ &= acx^{2d} + (ae + bc + ac + be - ac - be)x^d + be \\ &= acx^{2d} + ((a + b)(c + e) - ac - be)x^d + be \end{aligned}$$

So if we calculate ac, be and $\big((a + b)(c + e) - ac - be\big)$, multiply them by x^{2d}, x^d and 0 respectively and add them up, we can calculate the product of p_1 and p_2.

The products we need to calculate are ac, be and $(a + b)(c + e)$. Since add, subtract and shift takes $\theta(n)$ time, algorithm takes $T(n) = 3T\left(\frac{n}{2}\right) + \theta(n) = \theta\big(n^{\log_2 3}\big)$ time by master theorem.