Solutions to Exercises 3.1

4. a. Here is a pseudocode of the most straightforward version:

Algorithm *BruteForcePolynomialEvaluation*($P[0..n]$, x)
//The algorithm computes the value of polynomial P at a given point x
//by the "highest-to-lowest term" brute-force algorithm
//Input: Array $P[0..n]$ of the coefficients of a polynomial of degree n,
// stored from the lowest to the highest and a number x
//Output: The value of the polynomial at the point x
$p \leftarrow 0.0$
for $i \leftarrow n$ **downto** 0 **do**
  $power \leftarrow 1$
  for $j \leftarrow 1$ **to** i **do**
    $power \leftarrow power * x$
  $p \leftarrow p + P[i] * power$
return p

We will measure the input's size by the polynomial's degree n. The basic operation of this algorithm is a multiplication of two numbers; the number of multiplications $M(n)$ depends on the polynomial's degree only. Although it is not difficult to find the total number of multiplications in this algorithm, we can count just the number of multiplications in the algorithm's inner-most loop to find the algorithm's efficiency class:

$$M(n) = \sum_{i=0}^{n} \sum_{j=1}^{i} 1 = \sum_{i=0}^{n} i == \frac{n(n+1)}{2} \in \Theta(n^2).$$

b. The above algorithm is very inefficient: we recompute powers of x again and again as if there were no relationship among them. Thus, the obvious improvement is based on computing consecutive powers more efficiently. If we proceed from the highest term to the lowest, we could compute x^{i-1} by using x^i but this would require a division and hence a special treatment for $x = 0$. Alternatively, we can move from the lowest term to the highest and compute x^i by using x^{i-1}. Since the second alternative uses multiplications instead of divisions and does not require any special treatment for $x = 0$, it is both more efficient and cleaner. It leads to the following algorithm:

Algorithm *BetterBruteForcePolynomialEvaluation*($P[0..n]$, x)
//The algorithm computes the value of polynomial P at a given point x
//by the "lowest-to-highest term" algorithm
//Input: Array $P[0..n]$ of the coefficients of a polynomial of degree n,
// from the lowest to the highest, and a number x
//Output: The value of the polynomial at the point x
$p \leftarrow P[0]$; $power \leftarrow 1$
for $i \leftarrow 1$ **to** n **do**
  $power \leftarrow power * x$
  $p \leftarrow p + P[i] * power$
return p

The number of multiplications here is

$$M(n) = \sum_{i=1}^{n} 2 = 2n$$

(while the number of additions is n), i.e., we have a linear algorithm.

Note: Horner's Rule discussed in Section 6.5 needs only n multiplications (and n additions) to solve this problem.

Solutions to Exercises 3.2(簡答)

1. a. Cworst(n) = n + 1.
   b. Cavg(n) = (2−p)(n+1)

4. The algorithm will make 38 trials.

5. a. The total number of character comparisons will be c = 2∗ 9, 99, 996.
   b. The total number of character comparisons will be c = 4∗ 9, 99, 996.

6. The text composed of n zeros and the pattern $\underbrace{0 \ldots 0}_{m-1}1$ is an example of the worst-case input. The algorithm will make $m(n-m+1)$ character comparisons on such input.

Solutions to Exercises 3.4

2. The problem of solving traveling salesman problem is very similar to the finding of a Hamiltonian circuit. Generate all the solution of finding path from a particular city to all other city. Exhaustive search is a type of brute-force approach to solve combinatorial problems. It generates each and every combinatorial solution of the problem, selecting those which satisfy the problem's constraints and then finding a desired solution. Find the shortest cost tour through a given set of cities that a salesman visits in each city exactly once before returning to the city where he started.

4. Exhaustive search is a type of brute-force approach to solve combinatorial problems. It generates each and every combinatorial solution of the problem, selecting those which satisfy the problem's constraints and then finding a desired solution. Find the most valuable list of out-of n number of items that can easily fit into the knapsack.

5. Here is a very simple example: $\begin{bmatrix} 1 & 2 \\ 2 & 9 \end{bmatrix}$

6. Start by computing the sum S of the numbers given. If S is odd, stop because the problem doesn't have a solution. If S is even, generate the subsets until either a subset whose elements' sum is S/2 is encountered or no more subsets are left. Note that it will suffice to generate only subsets with no more than n/2 elements.