Sections 9.1, 9.2, 9.3

1) List three problems that have polynomial-time algorithms. Justify your answer.

1. The Single Source Shortest Path problem; i.e., find the shortest path between a given vertex and each other vertex in a weighted digraph, which may contain negative weights. There exists an O(V + E) algorithm that solves this problem (Bellman-Ford), where V is the number of vertices in the graph and E is the number of edges.
2. Standard linear programming problems: there exist generalized algorithms for solving them that have polynomial complexity (≈$O(n^4)$).
3. Finding the greatest common divisor of two integers: the Euclidian algorithm for solving this problem is polynomial in the size of the smaller input number.

2) Give a problem and two encoding schemes for its input. Express its performance using your encoding schemes.

Consider the sequence alignment algorithm (Algorithm 3.12) from Section 3.7:

▶ Algorithm 3.12 **Sequence Alignment Using Divide-and-Conquer**

Problem: Determine an optimal alignment of two homologous DNA sequences.

Inputs: A DNA sequence $\mathbf{x}$ of length m and a DNA sequence $\mathbf{y}$ of length n. The sequences are represented in arrays.

Outputs: The cost of an optimal alignment of the two sequences.

```
void opt (int i, int j)
{
    if (i == m)
        opt = 2(n - j);
    else if (j == n)
        opt = 2(m - i);
    else {
        if (x[i] == y[j])
            penalty = 0;
        else
            penalty = 1;
        opt = min(opt(i + 1, j + 1) + penalty, opt(i + 1, j) + 2,
                  opt(i, j + 1) + 2);
    }
}
```

The integers i, j, m, n are always represented on 32 bits, but we have two encoding schemes for the base characters (A, C, G, T):

1. 8-bit ASCII characters
2. 2-bit custom representation, with 4 bases stored in one Byte for storage efficiency

Because of the dynamic programming algorithm, the number of values of ***opt*** to be computed is bounded by m·n, and the most expensive integer manipulation performed for each of them is the assignment on the last line:

opt = min(opt(i + 1, j + 1) + penalty , opt(i + 1, j) + 2, opt(i , j + 1) + 2);

Denote by ***c*** the number of assembly instructions required to perform the operation above and all the integer operations (e.g. the test i == m), plus the stack operations required by the three recursive calls. The total number of steps is bounded by the expression

$$N^2(c + b),$$

Where N = max(n,m), and ***b*** is the nr. of steps required for the test x[i] == y[j] in the third if statement.

- For the encoding scheme #1, a commercial CPU will probably use two LOAD instructions to bring the array elements from RAM into CPU registers, followed by a BRE (branch if equal) or BNE (branch if not equal), for a total of 3 assembly instructions.
- For the encoding scheme #2, the CPU will additionally have to apply bit-level masks to the Bytes loaded, in order to extract the appropriate 2-bit representations of the bases; assuming for simplicity that the 4 possible masks are precomputed and stored in registers, this means two extra AND instructions, for a total of 5 assembly instructions.

In conclusion, encoding scheme #1 has a maximum nr. of $N^2(c + 3)$ steps, and encoding scheme #2 has a maximum nr. of $N^2(c + 5)$ steps; they are both $O(N^2)$.

3) 3. Show that a graph problem using the number of vertices as the measure of the size of an instance is polynomially equivalent to one using the number of edges as the measure of the size of an instance.

For a connected graph, the lower bound on the number of edges E in terms of the number of vertices V is V-1. The upper bound is V^2, a fully-connected graph. Thus, the number of edges is a polynomial function of the number of vertices, and the two input representations are polynomially equivalent.

4) In which of the three general categories discussed in Section 9.3 does the problem of computing the n^{th} Fibonacci term belong? Justify your answer.

Computing the n^{th} Fibonacci number is in P, since for any n, one only needs the values of the previous two numbers in the sequence. Thus, an iterative algorithm that only stores two numbers and computes the next number at each step (beginning with 0 and 1) can compute the n^{th} number in n steps. Since the complexity of the problem is based on the input size, the bound on such an algorithm is O(n log n), since it takes log n bits to store a number of size n and log n times steps to add two numbers of size n. The space complexity is thus O(log n).

5) A graph has an Euler Circuit if and only if (a) the graph is connected and (b) the degree of every vertex is even. Find a lower bound for the time complexity of all algorithms that determine if a graph has an Euler Circuit. In which of the three general categories discussed in Section 9.3 does this problem belong? Justify your answer.

Checking to see if a graph is connected can be performed in O(V+E) time using breadth-first search, where V is the number of vertices and E is the number of edges. Additionally, during this process the degree of each vertex can be recorded to verify whether it is even or not. Since each vertex must be visited and its degree determined by examining each of its outgoing edges, this constitutes a lower bound on the problem of determining whether a given graph has an Euler circuit. The problem is thus in P.

6) List at least two problems that belong in each of the three general categories discussed in Section 9.3.

- Polynomial-time algorithms have been found: $\Theta(n^2)$ algorithm for sequence alignment, Θ(nlg n) for the fractional knapsack.
- Proven intractable: generating all permutations od a string of n characters (unreasonable), Presburger Arithmetic (reasonable), Halting Problem (undecidable).
- Not proven intractable, and no polynomial algorithm found: finding longest path in a graph, 0-1 Knapsack where W cannot be used as input size parameter, deciding if a number n is prime.

Section 9.4

7) Implement the verification algorithm for the Traveling Salesperson Decision problem discussed in Section 9.4.1 on your system, and study its polynomial time performance.

```
#include <iostream>
#include <time.h>
using namespace std;

#define n 15
int W[n+1][n+1] = { { 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15},
                    { 1,  0, 29, 82, 46,  0,  0,  0,  0, 51, 55, 29, 74, 23, 72,  0},
                    { 2, 29,  0,  0,  0,  0,  0, 43, 23, 23, 31,  0, 51, 11, 52, 21},
                    { 3, 82,  0,  0,  0, 46, 55, 23, 43, 41, 29, 79, 21, 64,  0, 51},
                    { 4, 46, 46, 68,  0, 82, 15, 72, 31,  0,  0, 21, 51,  0,  0, 64},
                    { 5,  0,  0, 46, 82,  0,  0, 23, 52, 21, 46, 82, 58, 46, 65, 23},
                    { 6, 52, 43, 55, 15, 74,  0,  0,  0,  0,  0, 33, 37, 51, 29, 59},
                    { 7, 72, 43, 23, 72, 23, 61,  0, 42, 23, 31, 77, 37, 51, 46, 33},
                    { 8, 42, 23, 43, 31, 52, 23, 42,  0, 33,  0,  0,  0, 33,  0, 37},
                    { 9, 51, 23, 41, 62, 21, 55,  0,  0,  0, 29, 62, 46, 29, 51, 11},
                    {10, 55,  0, 29, 42, 46, 31, 31, 15, 29,  0, 51, 21, 41, 23, 37},
                    {11, 29, 41, 79, 21, 82, 33, 77,  0,  0, 51,  0, 65, 42,  0,  0},
                    {12,  0,  0,  0,  0, 58, 37,  0, 33, 46, 21, 65,  0, 61,  0,  0},
                    {13, 23, 11, 64,  0,  0,  0,  0,  0, 29, 41, 42, 61,  0, 62, 23},
                    {14,  2,  0,  0,  0, 65, 29, 46, 31, 51, 23, 59, 11, 62,  0,  0},
                    {15, 46,  0, 51,  0, 23, 59,  0,  0, 11,  0, 61,  0, 23, 59,  0}};
int S[n+2] = {0, 1, 2, 9, 4, 5, 7, 6, 3, 8, 15, 11, 12, 13, 10, 14, 1};
                    //claimed tour
#define D 500       //weight of tour not greater than this

bool visited(int key, int arr[n+1]){
       bool found = false;
       int i = 1;
       while (i<=n && !found)
              if (key == arr[i])
                     found = true;
              else
                     i++;
       return found;
}

bool verify(int w[][n+1], int d, int s[]){
       int sum = 0;
       int i = 1;
       bool OK_tour = true;
       int v[n+2] = {0};                //keeps track of nodes visited
       while (i<=n && OK_tour){
              if (w[s[i]][s[i+1]] <= 0){
                     OK_tour = false;
                     cout <<"Not a tour!";
                     cout <<"\tnode " <<s[i] <<" does not connect to node ";
                     cout <<s[i+1] <<endl;
              }
              else if (visited(s[i+1], v)){
                     OK_tour = false;
                     cout <<"Not a tour!";
                     cout <<"\tnode " <<s[i+1] <<" is a cycle"  <<endl;
              }
              else{
```

```
                        sum += w[s[i]][s[i+1]];
                        if (sum > d){
                                OK_tour = false;
                                cout <<"Weight too great: " <<sum <<endl;
                        }
                        else{
                                i++;
                                v[i] = s[i];
                                cout <<"node " <<s[i] <<" OK  ";
                                cout <<"sum = " <<sum <<endl;
                        }
                }
        }
        return OK_tour;
}


int main(){
        time_t begin, end;
        begin = time(NULL);

        if (verify(W, D, S)){
                cout <<"\nThis is a tour, and its weight is ";
                cout <<"no greater than " <<D <<endl;
        }

        end   = time(NULL);
        cout << "\ttime = " << difftime(end, begin) << " s" << endl;
        return 0;
}
```

Below are outputs obtained for d = 500, 600, and 700, respectively:

```
node 2 OK  sum = 29
node 9 OK  sum = 52
node 4 OK  sum = 114
node 5 OK  sum = 196
node 7 OK  sum = 219
node 6 OK  sum = 280
node 3 OK  sum = 335
node 8 OK  sum = 378
node 15 OK  sum = 415
node 11 OK  sum = 476
Weight too great: 541
        time = 0 s
```

```
node 2 OK  sum = 29
node 9 OK  sum = 52
node 4 OK  sum = 114
node 5 OK  sum = 196
node 7 OK  sum = 219
node 6 OK  sum = 280
node 3 OK  sum = 335
node 8 OK  sum = 378
node 15 OK  sum = 415
node 11 OK  sum = 476
node 12 OK  sum = 541
Weight too great: 602
        time = 0 s
```

```
node 2 OK  sum = 29
node 9 OK  sum = 52
node 4 OK  sum = 114
node 5 OK  sum = 196
node 7 OK  sum = 219
node 6 OK  sum = 280
node 3 OK  sum = 335
node 8 OK  sum = 378
node 15 OK  sum = 415
node 11 OK  sum = 476
node 12 OK  sum = 541
node 13 OK  sum = 602
node 10 OK  sum = 643
node 14 OK  sum = 666
node 1 OK  sum = 668

This is a tour, and its weight is no greater than 700
        time = 0 s
```

8) Establish that the problems in Examples 9.2 to 9.5 are in NP.

9.2 Traveling Salesperson: In the program from Exercise 7 above, function *verify* has a loop that executes n times in the worst case, calling each time the function *visited*, which also has a loop that executes n times in the worst case, therefore the verification algorithm is in $O(n^2)$.

9.3 0-1 Knapsack: Similar to the TP algorithm, this verification algorithm adds at most n weights, checking each time if the current item has not been already used, and checking at the end if the sum is ≥ P, therefore $O(n^2)$.

9.4 Graph-Coloring: In the worst case, the color of every vertex has to be compared with that of all the n-1 remaining vertices, therefore the verification algorithm is in $O(n^2)$.

9.5 Clique: To verify a clique of size k, the algorithm has to check if each of the k vertices has edges to each of the remaining k-1, therefore $O(k^2)$, and, since k ≤ n, also in $O(n^2)$.

9) A verification algorithm for the Clique Decision problem takes as input a graph G = (V,E), a subset of vertices U, and an integer k. The algorithm must return true if W is of size k and constitutes a clique, and return false or not halt otherwise.

```
boolean verifyClique(int W[][], int U[], int k) {
    if(U.size != k) return false;
    index i, j;
    for(i = 1; i < k; i++)
        for(j = 1; j < k; j++)
            if(W[U[i]][U[j]] == infinity) return false;
    return true;
}
```

The algorithm checks to see that the size of the input clique U is equal to k, and then checks each element of U to see that it is adjacent to each other element in U. If this is the case, the algorithm returns true; otherwise, it returns false. The algorithm has quadratic time complexity in the size of the clique U ($O(U^2)$), which is a subset of the input graph, and is thus polynomial in the size of the input.

10) Show that the reduction of the CNF-Satisfiability problem to the Clique Decision problem can be done in polynomial time.

Here is the relevant fragment from Example 9.9:

"Let

$$B = C1 \wedge C2 \wedge \cdots \wedge Ck$$

be a logical expression in CNF, where each Ci is a clause of B, and let x1,x2,...,xn be the variables in B. Transform B to a graph G = (V,E), as follows:

$$V = \{(y,i) \text{ such that } y \text{ is a literal in clause } Ci\}$$

$$E = \{((y,i),(z,j)) \text{ such that } i = j \text{ and } z = y\}.$$"

Since there are n literals, setting up the vertices in V takes $c_1 n$ steps, where c_1 is a constant.

There are at most $n(n-1)/2$ edges, and each takes a constant nr. of steps c_2 to set up.

The reduction algorithm therefore takes $c_1 n + c_2 n(n-1)/2 \leq 2cn^2$ steps, where $c = \max(c_1, c_2/2)$, so it is in $O(n^2)$.

11) Write a polynomial-time verification algorithm for the Hamiltonian Circuits Decision problem.

A verification algorithm for the Hamiltonian Circuits Decision problem takes as input a digraph G = (V,E), and a path P. The algorithm must return true if P begins and ends with the same vertex and visits each vertex exactly once, and return false or not halt otherwise.

```
boolean verifyHam(int n, int W[][], int P[]) {
    if(P.size != n+1) return false;
    if(P[1] != P[n+1]) return false;
    index i;
    for(i = 1; i <= n; i++)
        if(W[i][i+1] == infinity) return false;
    return true;
}
```

The algorithm checks to see that the path is of appropriate length and that it begins and ends with the same vertex. It then goes through the vertices of the path and checks to see that the sequence of vertices is a valid path. The first two checks are constant time operations, and the third is linear in the length of the input path P, which has an upper bound equal to the number of vertices in the graph (O(V)). The algorithm is thus polynomial in the size of the input.

12) Show that the reduction of the Hamiltonian Circuits Decision problem to the Traveling Salesperson (Undirected) Decision problem can be done in polynomial time.

Reducing the Hamiltonian Circuits Decision problem to the Traveling Salesman (Undirected) Decision problem involves creating a graph with the same number of vertices, V, as the input to the Hamiltonian Circuits problem, and adding an edge of appropriate weight (1 if the edge is in the original graph, and 2 otherwise) between every pair of vertices . The number of edges in a complete graph with V vertices is given by V^2, which is greater than or equal to the number of edges, E, in the original graph. Thus the complexity of the transformation is $O(V^2)$, which is polynomial in the size of the input.

13) Show that the reduction of the Traveling Salesperson (Undirected) Decision problem to the Traveling Salesperson Decision problem can be done in polynomial time.

Let V be the number of vertices in the graph. The number of edges in the undirected graph is $\leq V\cdot(V-1)/2 < V^2/2$, so the number of edges to be created in the directed graph is $< V^2$, thus the complexity of the transformation is $O(V^2)$, which is polynomial in the size of the input.

14) Show that a problem is NP-easy if and only if it [Turing-] reduces to an NP-complete problem.

Remark: From the remark right after the definition of NP-easy (p.425), it's clear that Turing-reduction is meant.

We prove the double implication:

i] Problem A is NP-easy $\Rightarrow$ A Turing-reduces to B (NP-complete)

Proof: A $\in$ NP-easy $\Rightarrow$ A α_T B $\in$NP, but, according to the definition of NP-complete, any NP problem, including B, reduces to any NP-complete problem, for example CNF: B α_T CNF. Since Turing-reduction implied many-to-one reduction, we have that B α_T CNF, so by transitivity A α_T CNF $\in$NP-complete.

ii] Problem A Turing-reduces to B (NP-complete) $\Rightarrow$ A is NP-easy

Proof: By definition, any NP-complete problem is also NP, so B is NP, so A Turing-reduces to B $\in$ NP.

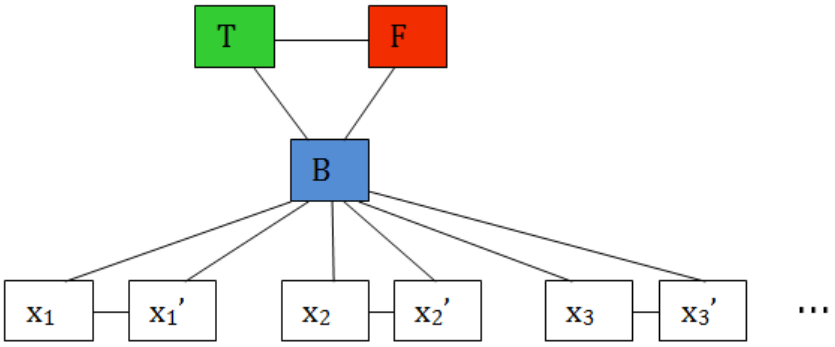
15) Suppose that problem A and problem B are two different decision problems. Furthermore, assume that problem A is polynomial-time many-one reducible to problem B. If problem A is NP-complete, is problem B NP-complete? Justify your answer.

The definition of NP-completeness (p.415) also requires for B to be in NP, which does not follow from our hypothesis. In fact, any decision problem A (in P or NP) can be trivially reduced to the halting problem by appending a *while (true)* loop right after the instruction that made the result true (or false), so B can be the halting problem, which is not in NP.

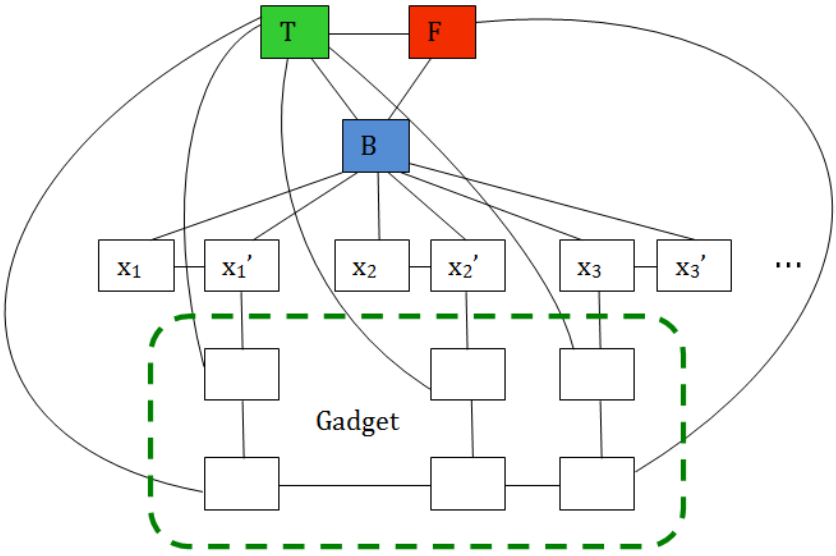
16) When all instances of the CNF-Satisfiability problem have exactly three literals per clause, it is called the 3-Satisfiability problem. Knowing that the 3-Satisfiability problem is NP-complete, show that the Graph 3-Coloring problem is also NP-complete.

Let's call the three colors T (True), F (False) and B (base). We describe how to construct a graph G(B) that is 3-Col if and only if the problem B = $C1 \wedge C2 \wedge \cdots \wedge Ck$ is 3-Sat:

- Create a "triangle" graph with 3 vertices T, F, and B, fully connected.
- Create pairs of vertices for each literal and its inverted version (e.g. x_1 and x_1'), and connect all of them to the B vertex.
- Connect together the vertices in each pair; this ensures that, in each pair, one vertex is T and the other F. The graph so far is shown below:



- For each clause C_i in B, we add to the graph above a group of 6 nodes, known in the literature as a "gadget". As an example, we shall assume that we have $C_1 = x_1' \vee x_2' \vee x_3$. The corresponding gadget is shown below in the dashed box:



- It is left to the reader to prove that the graph is 3-Colorable if and only if at least one literal in the clause is true.
- The gadget construction is repeated for every other clause in B.

17) Show that if a problem is not in NP, it is not NP-easy. Therefore, Presburger Arithmetic and the Halting problem are not NP-easy.

Reasoning by contradiction, let's assume that the problem A is not in NP, but it is NP-easy. This means that A has no polynomial-time verification algorithm, but it can be Turing-reduced to a decision problem B that does have a polynomial-time verification algorithm, i.e. to an easier problem – contradiction.

Section 9.5

18) Implement the approximation algorithms for the Traveling Salesperson problem, run them on your system, and study their performances using several problem instances.

Implementations and performances will vary.

19) Write a detailed algorithm of the approximation algorithm for the Bin-Packing problem given in Section 9.5.2, and show that its time complexity is in $\Theta(n^2)$.

```
int binPack(int n, number S[]) { //assumes S is sorted in nonincreasing order
        number B[1…n]; //initialized to zeroes
        index i, j, k = 1;
        for(i=1; i <= n; i++) {
                boolean packed = false;
                for(j=1; j <= k; j++)
                        if(1-B[j] >= S[i]) {
                                B[j] = S[i];
                                packed = true;
                        }
                if(!packed) {
                        k++;
                        B[k] = S[i];
                }
        }
        return k;
    }
```

For each item, the algorithm searches through the bins that have already been opened to find a bin into which to place the item. In the worst case, each item must be placed in a separate bin, and thus at each iteration all of the open bins must be checked before opening a new one. The time complexity of this worst-case scenario is thus equal to the sum of the integers from 1 to n, which is $n(n+1)/2$; therefore, the algorithm has $\Theta(n^2)$ complexity.

20) For the Sum-of-Subsets problem discussed in Chapter 5, can you develop an approximation algorithm that runs in polynomial time?

As explained in Section 5.4, the Sum-of-Subsets is a particular case of the Knapsack problem, when we just want to find a subset of objects whose total weight equals the capacity W (irrespective of their value, or assuming they all have the same value per unit of weight). There, the problem was solved with backtracking, but here we use a different idea:

- Let's start with a list containing just the number zero: $L_0 = [0]$.
- We define the following operation to be applied to a list L: f(L) is another list, obtained by:
    - Taking all the distinct sums between an element of L and an element of the array of weights w: L[i] + w[j]
        - Note that there can be length(L)·length(w) = m·n such sums in the worst case.
    - Taking the union between L and the sums above
        - The union can contain (m+1)n numbers in the worst case
    - Sorting the new list
    - Dropping the sums that are > W.
        - There can still be (m+1)n numbers in the worst case
- If we apply the operation f to L_0 n times, we obtain a sequence of lists L_1, …, L_n,
- The last of them, L_n, contains all the candidates
    - There can be $\Omega(n^n)$ numbers in L_n in the worst case (we neglected the "+1")
- The max(L_n) is the solution.

The problem with the algorithm above is that it is worse than exponential in n, however, we can do better if all we need is an approximation:

Instead of keeping all the sums < W at each step, we only keep those that are "about" a small constant ε from each other. One way to accomplish this is to split the interval [0..W] into bins of size ε and only retain one number in each bin. This guarantees that the least at each step has no more than W/ε elements, avoiding the combinatorial explosion.

It can be easily proved that, by doing this, the error we make only increases by at most ε at each step, so it is bounded by nε for the entire algorithm. If we have a target error E for the entire algorithm, we choose ε = E/n, so the length of the list is at most Wn/E at each step, so the additions and sorting take polynomial time at each step, therefore polynomial time for the n iterations.

21) Can you develop an approximation algorithm for the CNF-Satisfiability problem by stating it as an optimization problem—that is, by finding a truth assignment of the literals in the expression that makes the maximum possible number of clauses true?

Let's call S_1 the assignment of True values to all the literals, and S_2 the assignment of False values to all literals. We then choose between S_1 and S_2 the one that makes the most clauses True. In the worst case, no clauses have mixed inverted and non-inverted literals (i.e. in a clause the literals are either all inverted or all non-inverted), and therefore the numbers of True clauses in $S_1$and S_2 add up to n. This proves that the algorithm makes at least 50% of the clauses true.

Remark: A better polynomial algorithm exists (Karloff-Zwick), that satisfies at least 7/8 of the clauses.

Additional Exercises

22) Can an algorithm be a polynomial-time algorithm for a problem using one encoding scheme and be an exponential-time algorithm for the same problem using another encoding scheme? Justify your answer.

Yes. Consider the case of an O(n) algorithm in which the input is represented in binary. The same algorithm given the same input in unary would be $O(2^n)$, since a number n in binary takes 2^n symbols to write in unary.

23) Write a verification algorithm for the composite number problem (Example 9.14) and analyze it to show that it is a polynomial-time algorithm.

The verification algorithm is simply the long division algorithm from grade school. If we divide a/b, its complexity is bounded by n (nr. of digits of a) times m (nr. of digits of b), O(nm): there are at most n-m steps (round it up to n, assuming n >>m), and at each step we have to multiply the m-digit number b by the current digit q_i found in the quotient, then subtract bq_i, which is O(m).

24)
```
boolean composite(int n) {
    int i = 2;
    while(i <= floor(n^0.5)) {
        if(n % i == 0)
            return true;
        i++;
    }
    return false;
}
```

The algorithm goes through at most floor($n^{0.5}$) iterations of the while loop, and the modulus operation takes $O(\log^2 n)$ time (division of two numbers each with log n bits). The algorithm is thus $O(n^{0.5} \log^2 n)$, a polynomial algorithm.

25) Is the Towers of Hanoi problem an NP-complete problem? Is it an NP-easy problem? Is it an NP-hard problem? Is it an NP-equivalent problem? Justify your answers. This problem is presented in Exercise 17 in Chapter 2.

It is not NP, because the solution has $2^n - 1$ moves, and in the worst case the verification algorithm has to examine all of them.

It is not NP-easy (because NP-easy requires NP), and therefore not NP-equivalent.

It is (trivially) NP-hard, because the definition of NP-hard is "if a polynomial algorithm did exist …". Since we know that a polynomial algorithm does not exist for ToH, the hypothesis is false, and therefore it can imply anything.

26) Given a list of n positive integers, divide the list into two sublists such that the absolute value of difference between the sums of the integers in the two sublists is minimized. Is this problem an NP-complete problem? Is this problem an NP-hard problem?

This is known as the ***Partition Problem***. As a decision problem (call it A), it requires to find out if the list can be partitioned such that the difference is 0, and it has been proved to be NP-complete, however, the problem as stated in the Exercise (call it B) is not a decision problem, and therefore cannot be NP.

If we had a polynomial-time solution to B, A could be solved easily (in $\Theta(1)$ steps) by simply examining the solution and returning True if the difference is zero and False if it is not. Therefore the NP-complete problem A is Turing-reducible to B, so B is NP-hard.

A polynomial-time approximation algorithm exists, similar to the ***nonincreasing first fit*** presented in the text for the bin packing problem: Iterate through the numbers in nonincreasing order, assigning then to the set currently having the smaller sum. The complexity is $\Theta(n \lg n)$ due to sorting, and it can be proved that the result is at the most 4/3 of the optimal solution.