Sections 5.1, 5.2

1) n=6:

	Q				
			Q		
					Q
Q					
		Q			
				Q	

		Q			
					Q
	Q				
				Q	
Q					
			Q		

n=7:

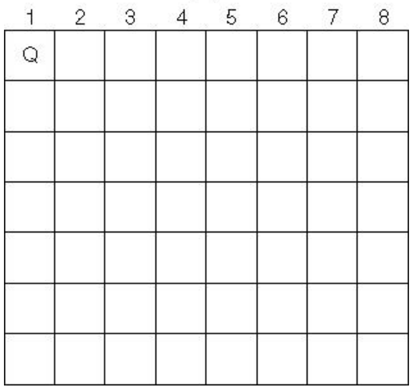
Q						
		Q				
				Q		
						Q
	Q					
			Q			
					Q	

Q						
			Q			
						Q
		Q				
					Q	
	Q					
				Q		

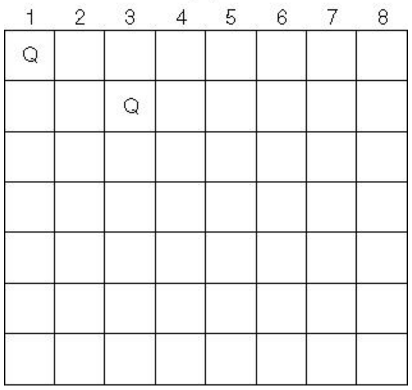
2) Apply the Backtracking algorithm for then-Queens problem (Algorithm 5.1) to the problem instance in which n = 8, and show the actions step by step. Draw the pruned state space tree produced by this algorithm up to the point where the first solution is found.

Solution:

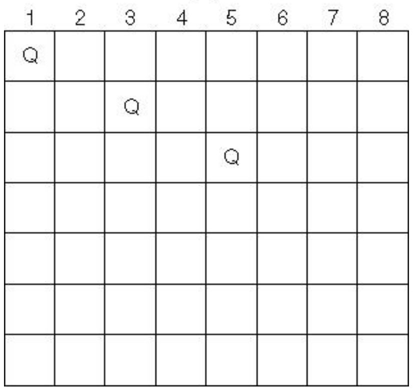
(1)



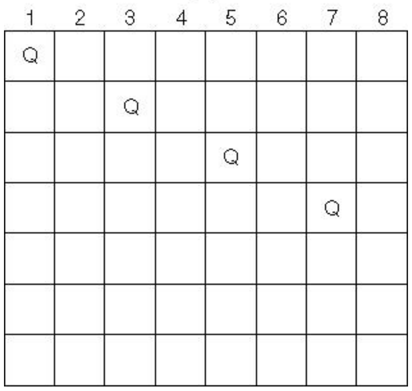
(2)



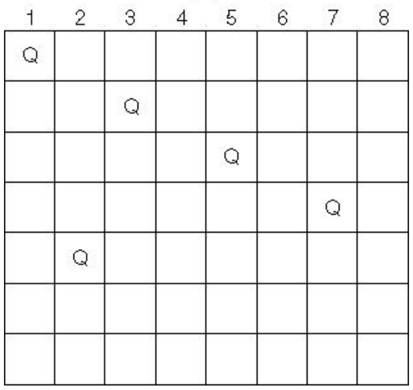
(3)



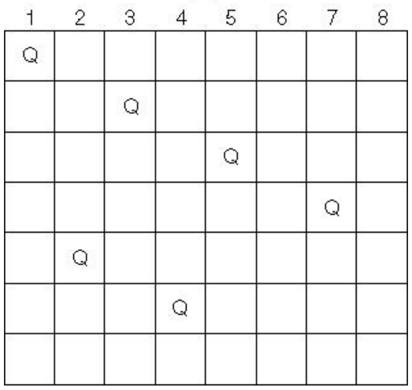
(4)

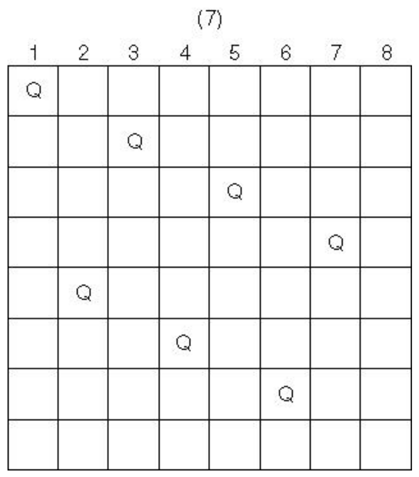


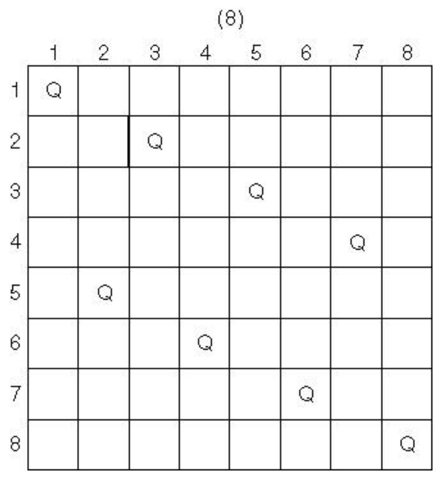
(5)



(6)







Solution {1, 3, 5, 7, 2, 4, 6, 8}

3) Problem: Position n queens on an n × n chessboard so that no two queens threaten each other.

Input: positive integer n

Output: All possible ways in which n queens can be placed on an n×n chessboard so that no two queens threaten each other. Each output consists of an array of integers col indexed from 1 to n, where col[i] is the column in where the queen in the ith row is placed.

```
void queens(index i) {
            index j;
      if(i==n)
            cout << col[1] through col[n];
      else
            for (j=1; j <= n; j++) {
                  col[i+1] = j;
                  if (promising(i+1))
                        queens(i+1);
            }
}
```

Initial call is queens(0). The promising function is identical to the one on page 212.

4) Problem: Find the number of ways in which n queens can be placed on an n×n chessboard so that no two queens threaten each other.

Input: positive integer n

Output: Number of ways in which n queens can be placed on an n×n chessboard so that no two queens threaten each other.

```
        void queens(index i, int& count) {
                    index j;
        if(i==n)
                    count++;
        else
                    for(j=1; j <= n; j++) {
                              col[i+1] = j;
                              if(promising(i+1))
                                        queens(i+1);
}
}
```

With an integer variable count declared and initialized to 0, initial call is queens(0, &count). The promising function is identical to the one on page 212.

5) Show that, without backtracking, 155 nodes must be checked before the first solution to the n =4 instance of then-Queens problem is found (in contrast to the 27 nodes in Figure 5.4).

Solution: Without backtracking, we have to also visit all the subtrees of the “x”-ed nodes from Fig.5.4:

The 5 nodes on level 2 have 5 x 4 x 4 =80 nodes.

The 7 nodes on level 3 have 7 x 4 = 28 nodes.

Total = 80 + 28 = 108 extra nodes.

Including the 27 from backtracking, we have **135** total nodes.

6) Implement the Backtracking algorithm for the n-Queens problem (Algorithm 5.1) on your system, and run it on problem instances in which n = 4, 8, 10, and 12.

Solution: The code is the one in Algorithm 5.1.

n = 4	:	2 solutions
n = 8	:	92 solutions
n = 10	:	724 solutions
n = 12	:	14, 200 solutions

7) Improve the Backtracking algorithm for the n-Queens problem (Algorithm 5.1) by having the promising function keep track of the set of columns, of left diagonals, and of right diagonals controlled by the queens already placed.

Solution: The set of columns can be an array **`col_used[n+1]`**, with zeroes for the columns in use and ones for the columns not in use. Similarly, since there are 2n-1 left diagonals and 2n-1 right diagonals (anti-diagonals), these can be stored, respectively, in arrays **`left_diag_used[2n]`** and **`right_diag_used[2n]`**.

8) Modify the Backtracking algorithm for then-Queens problem (Algorithm 5.1) so that, instead of generating all possible solutions, it finds only a single solution.

Solution: Insert a **return** statement in the queens() function after the solution is displayed:

```
if (promising(i))
    if (i == n) {
        cout << col [1] through col [n];
        return;
    }
    else ....
```

9) Suppose we have a solution to the n-Queens problem instance in which n = 4. Can we extend this solution to find a solution to the problem instance in which n = 5? Can we then use the solutions for n = 4 and n = 5 to construct a solution to the instance in which n=6 and continue this dynamic programming approach to find a solution to any instance in which n> 4?

Justify your answer.

<u>Solution</u>: for n = 4 queens, there are only two solutions (for each column from left to right, the row where the queen is placed is shown):

1. $\boxed{(2, 4, 1, 3)}$ 2. $\boxed{(3, 1, 4, 2)}$

For n= 5 queens, there are 10 solutions, and two of them contain as sub-solutions the solutions from n = 4:

1. (1, 3, 5, 2, 4) 2. (1, 4, 2, 5,3)

3. $\boxed{(\mathbf{2, 4, 1, 3}, 5)}$ 4. (2, 5, 3, 1, 4)

5. $\boxed{(\mathbf{3, 1, 4, 2}, 5)}$ 6. (3, 5, 2, 4, 1)

7. (4, 1, 3, 5, 2) 8. (4, 2, 5, 3, 1)

9. (5, 2, 4, 1, 3) 10. (5, 3, 1, 4, 2)

For n= 6 queens, there are only 4 solutions, and none of them contains as sub-solutions the solutions from n = 4 or n = 5:

1. (2, 4, 6, 1, 3, 5) 2. (3, 6, 2, 5, 1, 4)

3. (4, 1, 5, 2, 6, 3) 4. (5, 3, 1, 6, 4, 2)

In conclusion, this approach does not work for general n.

The only known dynamic programming solution to the n-Queens problem employs more advanced concepts. It was published as

I. Rivin and R. Zabih, *A dynamic programming solution to the n-queens problem*, Information Processing Letters 41, pp.253-256, 1992.

10) There are no solutions to the n queens problem when n=2 or n=3.

Section 5.3

11) Implement algorithm 5.3 (Monte Carlo estimate for the Backtracking algorithm for then-Queens problem) on your system, run it 20 times on the problem instance in whichn = 8, and find the average of the 20 estimates.

Solution: Promising nodes can be found using

$1 + n + n(n-1) + n(n-1)(n-2) + \dots + n!$

Total number of nodes checked using

$1 + 1 \times t_0 + m_0 t_1 + m_0 m_1 t_2 + m_0 m_1 m_2 t_3 + m_0 m_1 \dots m_{i-1} t_i$

where t_i = total number of children at level i

m_i = no of promising children for a randomly selected node at level;

The code is an implementation of Algorithm 5.3. Being a probabilistic algorithm, the estimates will vary.

12) Modify the Backtracking algorithm for the n-Queens problem (Algorithm 5.1) so that it finds the number of nodes checked for an instance of a problem, run it on the problem instance in which n = 8, and compare the result against the average of Exercise 11.

Solution: We create a global variable ***int node_counter = 0;***, which we increment in the for loop of Algorithm 5.1 thus:

```
for (j = 1; j <= n; j++)  {
        col [ i + 1] = j ;
        queens(i + 1);
        node_counter++;
}
```

The number of nodes checked is 15,721 when n = 8, as listed in Table 5.1.

Section 5.4

13) Use the Backtracking algorithm for the Sum-of-Subsets problem (Algorithm 5.4) to find all combinations of the following numbers that sum to W =52:

w1 =2 w2 =10 w3 =13 w4 =17 w5 =22 w6 =42

Show the actions step by step.

Solution: As specified in the text, we assume that the indices range 1 through n. The weights are already sorted. The following presents the first 4 levels of recursion (out of the total of 6). Each innermost node below has to be further split twice (unless a node evaluates not promising):

0. Start from the non-existent w0, weight=0 promising → include[1] = yes
   a. weight=2, w1 promising → include[2] = yes
      i. weight=12, w2 promising → include[3] = yes
         1. weight=25, w3 promising → include[4] = yes
         2. weight=25, w3 promising → include[4] = no
      ii. weight=12, w2 is promising → include[3] = no
         1. weight=12, w3 promising → include[4] = yes
         2. weight=12, w3 promising → include[4] = no
   b. weight=2, w1 promising → include[2] = no
      i. weight=2, w2 promising → include[3] = yes
         1. weight=15, w3 promising → include[4] = yes
         2. weight=15, w3 promising → include[4] = no
      ii. weight=2, w2 promising → include[3] = no
         1. weight=2, w3 promising → include[4] = yes
         2. weight=2, w3 promising → include[4] = no
1. Back to the non-existent w[0], weight=0, promising → include[1] = no
   a. weight=0, w1 promising → include[2] = yes
      i. weight=10, w2 promising → include[3] = yes
         1. weight=23, w3 promising → include[4] = yes
         2. weight=23, w3 promising → include[4] = no
      ii. weight=10, w2 promising → include[3] = no
         1. weight=10, w3 promising → include[4] = yes
         2. weight=10, w3 promising → include[4] = no
   b. weight=0, w1 promising → include[2] = no
      i. weight=0, w2 promising → include[3] = yes
         1. weight=13, w3 promising → include[4] = yes
         2. weight=13, w3 promising → include[4] = no
      ii. weight=0, w2 promising → include[3] = no
         1. weight=0, w3 promising → include[4] = yes
         2. weight=0, w3 promising → include[4] = no

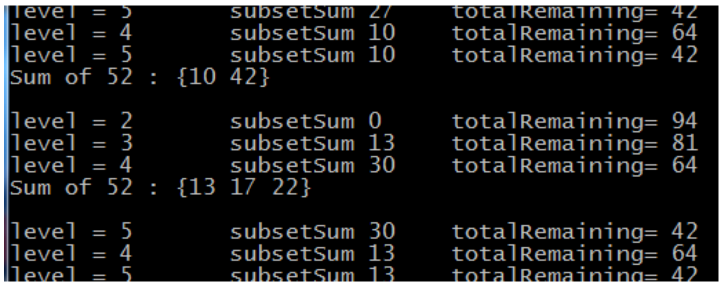
There are two solutions: {10, 42} and {13, 17, 22}.

14) Implement the Backtracking algorithm for the Sum-of-Subsets problem (Algorithm 5.4) on your system, and run it on the problem instance of Exercise 13.

Solution:

```
#include <iostream>
using namespace std;

int n = 6;
// The index 0 is not being used, to be consistent with Alg.5.4
int weights[] = {-1, 2, 10, 13, 17, 22, 42};
bool include[] = {false, false, false, false, false, false, false};
int W = 52;

void printSolutionSubset() {
    cout << "Sum of " << W << " : {";
    for (int i=1; i<=n; i++) {
        if (include[i])
            cout << weights[i] << " ";
    }
    cout << "\b}" << endl << endl;
}

//TORI = totalOfRemainingItems
bool promising(int level, int subsetSum, int TORI) {
    if (true){
        if (subsetSum == W)
            return true;
        if (subsetSum > W ||
            (subsetSum + TORI < W) ||
            (subsetSum+weights[level] > W))
            return false;
        else
            return true;
    }
}
```

```
void sumOfSubsets (int level, int subsetSum, int TORI) {
    cout << "level = " << level << "\tsubsetSum "<<subsetSum;
    cout <<"\ttotalRemaining= " << TORI << endl;
    if (level >= n)
        return;
    if (promising(level+1,subsetSum+weights[level+1],TORI-weights[level+1])) {
        include[level+ 1] = true;
        if (subsetSum + weights[level+1] == W)      // solution found
            printSolutionSubset();
        else
            sumOfSubsets(level+1, subsetSum + weights[level+1], TORI-weights[level+1]);
    }
    include[level+1] = false;
    if (promising(level+1, subsetSum, TORI-weights[level+1]))
        sumOfSubsets(level+1, subsetSum, TORI-weights[level+1]);
}

int main() {
    int total = 0;
    for (int i=1; i<=n; i++)
        total += weights[i];
    sumOfSubsets(0, 0, total);
    return 0;
}
```



```
level = 5          subsetSum 27       totalRemaining= 42
level = 4          subsetSum 10       totalRemaining= 64
level = 5          subsetSum 10       totalRemaining= 42
Sum of 52 : {10 42}

level = 2          subsetSum 0        totalRemaining= 94
level = 3          subsetSum 13       totalRemaining= 81
level = 4          subsetSum 30       totalRemaining= 64
Sum of 52 : {13 17 22}

level = 5          subsetSum 30       totalRemaining= 42
level = 4          subsetSum 13       totalRemaining= 64
level = 5          subsetSum 13       totalRemaining= 42
```

15) Write a backtracking algorithm for the Sum-of-Subsets problem that does not sort the weights in advance. Compare the performance of this algorithm with that of Algorithm 5.4.

Solution: The only difference from Algorithm 5.4 is that the **promising()** function cannot use the condition **weight + w[i + 1] <= W**; the array not being sorted, it is possible for the sum to exceed W for i+1, but still be ≤ W for a subsequent index. The new code for is:

```
bool promising ( index i ) {
    return (weight + total >= W) && (weight == W) ;
}
```

This algorithm is slower than Algorithm 5.4, because the absence of the extra condition in *promising()* causes it to explore more “dead-end” subtrees.

16)
```
void sum_of_subsets(index i, int weight, int total) {
        if(promising(i)) {
                if (weight==W) {
                        cout << include[1] through include[i];
                        return;
                }
                else {
                        include[i+1] = “yes”;
                        sum_of_subsets(i+1, weight + w[i+1], total – w[i+1]);
                        include[i+1] = “no”;
                        sum_of_subsets(i+1, weight, total – w[i+1]);
                }
        }
}
```

Promising function is the same as the one on page 224.

The worst-case complexity of this algorithm is the same as that of Algorithm 5.4, since in the event of no solution the same number of nodes must be checked. Even if there is a solution, problem instances may be constructed that have only one solution but still require the visitation of exponentially many nodes in the state-space tree.

17) Use the Monte Carlo technique to estimate the efficiency of the Backtracking algorithm for the Sum-of-Subsets problem (Algorithm 5.4).

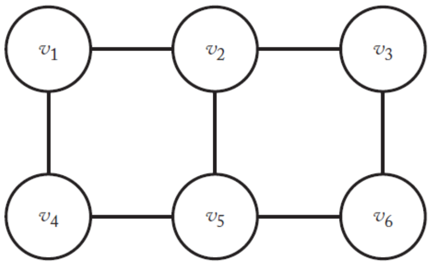
Solution: Use Algorithm 5.3 (Monte Carlo for n-Queens) as template, and adapt it to implement the data types and data structures from Algorithm 5.4 (Sum-of-Subsets). Only the minimal changes (indicated in **boldface**) were made in the code below:

```
int estimate_sum_subsets ( int n) {
        index i
        bool j;
        bool include[1…n];            //solution is an array of yes/no
        int m, mprod , numnodes ;
        set_of_index prom_children ;

        i = 0;                                  //root
        numnodes = 1;
        m = 1;                                  //we assume the root is promising
        mprod = 1;
        while (m != 0 && i !=n) {
                mprod = mprod * m;
                numnodes = numnodes + mprod * n ;
                i++;                            //next level in the tree
                m = 0 ;                         //assume no promising children
                prom_children = ∅;             //initialize with empty set
                for ( j in {True, False}) {
                        include[ i ] = j;
                        i f ( promising ( i ) )  {
                                m++;
                                prom_children = prom_children ∪ { j};
                        }
                }
                i f (m != 0) {
                        j = random selection from prom_children ;
                        include[ i ] = j ;
                }
        }
        return numnodes ;
}
```

The *promising* function is the same as in Algorithm 5.4.

Section 5.5

18) Use the Backtracking algorithm for the m-Coloring problem (Algorithm 5.5) to find all possible colorings of the graph below using the three colors red, green, and white. Show the actions step by step.



<u>Solution</u>:

- v1 red → promising
  - v2 red → not promising
  - v2 gre → promising
    - v3 red → promising
      - v4 red → not promising
      - v4 gre → promising
        - v5 red → promising
          - v6 red → not promising
          - **v6 gre → promising**
          - **v6 whi → promising**
        - v5 gre → not promising
        - v5 whi → promising
          - v6 red → not promising
          - **v6 gre → promising**
          - v6 whi → not promising
      - v4 whi → promising
        - v5 red → promising
          - v6 red → not promising
          - **v6 gre → promising**
          - **v6 whi → promising**
        - v5 gre → not promising
        - v5 whi → not promising
    - v3 gre → not promising
    - v3 whi → promising
      - v4 red → not promising
      - v4 gre → promising
        - v5 red → promising
          - v6 red → not promising
          - **v6 gre → promising**
          - v6 whi → not promising

      - v5 gre → not promising
      - v5 whi → promising
        - **v6 red → promising**
        - **v6 gre → promising**
        - v6 whi → not promising
    - v4 whi → promising
      - v5 red → promising
        - v6 red → not promising
        - **v6 gre → promising**
        - v6 whi → not promising
      - v5 gre → not promising
      - v5 whi → not promising
- v2 whi → promising
  - v3 red → promising
    - v4 red → not promising
    - v4 gre → promising
      - v5 red → promising
        - v6 red → not promising
        - **v6 gre → promising**
        - **v6 whi → promising**
      - v5 gre → not promising
      - v5 whi → not promising
    - v4 whi → promising
      - v5 red → promising
        - v6 red → not promising
        - **v6 gre → promising**
        - **v6 whi → promising**
      - v5 gre → promising
        - v6 red → not promising
        - v6 gre → not promising
        - **v6 whi → promising**
      - v5 whi → not promising
  - v3 gre → promising
    - v4 red → not promising
    - v4 gre → promising
      - v5 red → promising
        - v6 red → not promising
        - v6 gre → not promising
        - **v6 whi → promising**
      - v5 gre → not promising
      - v5 whi → not promising
    - v4 whi → promising
      - v5 red → promising
        - v6 red → not promising
        - v6 gre → not promising
        - **v6 whi → promising**

```
                v5 gre → promising
                    v6 red → not promising
                    v6 gre → promising
                    v6 whi → promising
                v5 whi → not promising
        v3 whi → not promising
v1 gre → promising
    v2 red → promising
        v3 red → not promising
        v3 gre → promising
            v4 red → promising
                v5 red → not promising
                v5 gre → promising
                    v6 red → promising
                    v6 gre → not promising
                    v6 whi → promising
                v5 whi → promising
                    v6 red → promising
                    v6 gre → not promising
                    v6 whi → not promising
            v4 gre → not promising
            v4 whi → promising
                v5 red → not promising
                v5 gre → promising
                    v6 red → promising
                    v6 gre → not promising
                    v6 whi → promising
                v5 whi → not promising
        v3 whi → promising
            v4 red → promising
                v5 red → not promising
                v5 gre → promising
                    v6 red → promising
                    v6 gre → not promising
                    v6 whi → not promising
                v5 whi → promising
                    v6 red → promising
                    v6 gre → promising
                    v6 whi → not promising
            v4 gre → not promising
            v4 whi → promising
                v5 red → not promising
                v5 gre → promising
                    v6 red → promising
                    v6 gre → not promising
                    v6 whi → not promising
```

                v5 whi → not promising
    v2 gre → not promising
    v2 whi → promising
        v3 red → promising
            v4 red → promising
                v5 red → not promising
                v5 gre → promising
                    v6 red → not promising
                    v6 gre → not promising
                    v6 whi → promising
                v5 whi → not promising
            v4 gre → not promising
            v4 whi → promising
                v5 red → promising
                    v6 red → not promising
                    v6 gre → promising
                    v6 whi → promising
                v5 gre → promising
                    v6 red → not promising
                    v6 gre → not promising
                    v6 whi → promising
                v5 whi → not promising
        v3 gre → promising
            v4 red → promising
                v5 red → not promising
                v5 gre → promising
                    v6 red → promising
                    v6 gre → not promising
                    v6 whi → promising
                v5 whi → not promising
            v4 gre → not promising
            v4 whi → promising
                v5 red → promising
                    v6 red → not promising
                    v6 gre → not promising
                    v6 whi → promising
                v5 gre → promising
                    v6 red → promising
                    v6 gre → not promising
                    v6 whi → promising
                v5 whi → not promising
        v3 whi → not promising
<u>v1 whi</u> → promising
    v2 red → promising
        v3 red → not promising
        v3 gre → promising

- v4 red → promising
    - v5 red → not promising
    - v5 gre → promising
        - **v6 red→ promising**
        - v6 gre → not promising
        - **v6 whi → promising**
    - v5 whi→ promising
        - **v6 red→ promising**
        - v6 gre → not promising
        - v6 whi → not promising
- v4 gre → promising
    - v5 red → not promising
    - v5 gre → not promising
    - v5 whi→ promising
        - **v6 red→ promising**
        - v6 gre → not promising
        - v6 whi → not promising
- v4 whi→ not promising

- v3 whi→ promising
    - v4 red → promising
        - v5 red → not promising
        - v5 gre → promising
            - **v6 red→ promising**
            - v6 gre → not promising
            - v6 whi → not promising
        - v5 whi→ promising
            - **v6 red→ promising**
            - **v6 gre→ promising**
            - v6 whi → not promising
    - v4 gre → promising
        - v5 red → not promising
        - v5 gre → not promising
        - v5 whi→ promising
            - **v6 red→ promising**
            - **v6 gre→ promising**
            - v6 whi → promising
    - v4 whi→ not promising

- v2 gre → promising
    - v3 red → promising
        - v4 red → promising
            - v5 red → not promising
            - v5 gre → not promising
            - v5 whi→ promising
                - v6 red → not promising
                - **v6 gre→ promising**
                - v6 whi → not promising

    - v4 gre → promising
      - v5 red → promising
        - v6 red → not promising
        - **v6 gre → promising**
        - **v6 whi → promising**
      - v5 gre → not promising
      - v5 whi → promising
        - v6 red → not promising
        - **v6 gre → promising**
        - v6 whi → not promising
    - v4 whi → not promising
  - v3 gre → not promising
  - v3 whi → promising
    - v4 red → promising
      - v5 red → not promising
      - v5 gre → not promising
      - v5 whi → promising
        - **v6 red → promising**
        - **v6 gre → promising**
        - v6 whi → not promising
    - v4 gre → promising
      - v5 red → promising
        - v6 red → not promising
        - **v6 gre → promising**
        - v6 whi → not promising
      - v5 gre → not promising
      - v5 whi → promising
        - **v6 red → promising**
        - **v6 gre → promising**
        - v6 whi → not promising
    - v4 whi → not promising
- v2 whi → not promising

There are 54 solutions. We can read them off starting from a bold leaf and working our way up the tree; for example, the last bold leaf above represents the solution (underlined):

Vertex	**Color**
v1	white
v2	green
v3	white
v4	green
v5	white
v6	green

19) Suppose that to color a graph properly we choose a starting vertex and a color to color as many vertices as possible. Then we select a new color and a new uncolored vertex to color as many more vertices as possible. We repeat this process until all the vertices of the graph are colored or all the colors are exhausted. Write an algorithm for this greedy approach to color a graph of *n* vertices. Analyze this algorithm and show the results using order notation.

Solution:

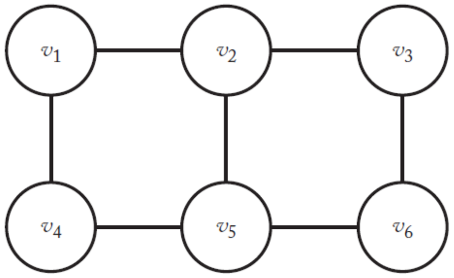
Inputs: positive integers ***n*** and ***m***, and an undirected graph containing *n* vertices. The graph is represented by a two-dimensional array ***W***, which has both its rows and columns indexed from 1 to n, where W[i]j] is true if there is an edge between ith vertex and the jth vertex and false otherwise.

Outputs: An m-coloring of the graph, as an array ***vcolor*** indexed from 1 to n (vcolor [i] is the color - integer between 1 and m - assigned to vertex i). In case the algorithm fails to find a solution, no array *vcolor* is printed. As in Algorithm 5.5, we assume that *vcolor* is a global array.

```
void m_coloring_greedy (int n, int m) {
        set_of_colored_nodes = ∅;                           //no nodes are colored yet
        for (int color = 1; color <= m, color++) {          //cycle through all colors
                for (int i not in set_of_colored_nodes)     //cycle through all nodes of graph
                        if(promising (i, color)) {
                                set_of_colored_nodes = set_of_colored_nodes ∪ {i};
                                vcolor(i) = color;
                                if(set_of_colored_nodes is full) {
                                        cout << vcolor[1] through vcolor[n]; //found solution
                                        return;             //exit function
                                }

                        }
        }
}

bool promising ( int i, int color) {
        bool switch = true;
        for (int j  in set_of_colored_nodes){          //Check if adjacent vertex is this color
                if (W[i][j] && vcolor[i]==vcolor[j])
                switch=false;
        }
        return switch;
}
```

Complexity analysis: *promising()* is $\Theta(n)$, and it is called inside the double loop of m_coloring_greedy, which executes n x m times → $\Theta(n^2m)$ overall.

20) m-Coloring using the above algorithm. The order of the colors is red, green, white.



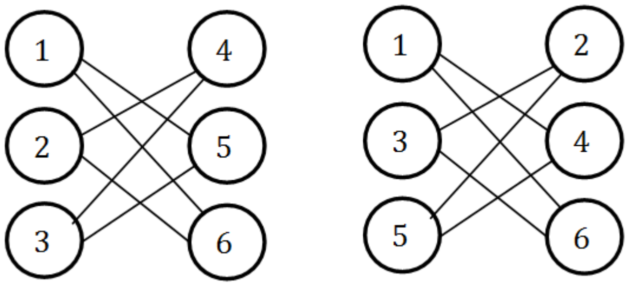
Solution:

Step 1 → color = red → nodes 1, 3, 5 are colored red

Step 2 → color = green → nodes 2, 4, 6 are colored green

21) Suppose we are interested in minimizing the number of colors used in coloring a graph. Does the greedy approach of Exercise 17 guarantee an optimal solution? Justify your answer.

Solution: No, the algorithm does not always use the minimum number of colors, because it is sensitive to the **ordering** of the vertices. For instance, the graph on the left is colored with two colors, but the one on the right requires three colors:



22) Any graph in which node n has an edge to every other node in the graph, and nodes n-2, n-1, and n form a clique. The graph given by the following adjacency matrix is one example.

0	0	0	0	0	1
0	0	0	0	0	1
0	0	0	0	0	1
0	0	0	0	1	1
0	0	0	1	0	1
1	1	1	1	1	0

23) Compare the performance of the Backtracking algorithm for the m-Coloring problem (Algorithm 5.5) and the greedy algorithm of Exercise 19. Considering the result(s) of the comparison and your answer to Exercise 19, why might one be interested in using an algorithm based on the greedy approach?

Solution: The greedy algorithm is more efficient ($\Theta(n^2m)$) than the Backtracking algorithm ($\Theta(m^n)$), so for large n it may be worth finding a non-optimal solution fast.

Besides, there are simple heuristics that can greatly improve the performance of the greedy algorithm at little extra cost, like the Welsh-Powell algorithm (in which the vertices are sorted in decreasing (non-increasing) order of their degrees, so the added cost is only $\Theta(n^2)$ or $\Theta(n\cdot\lg n)$.

24) Problem: Determine all ways in which all n vertices of an undirected graph can be colored using only two colors so that adjacent vertices are not the same color.

Input: positive integer n and an undirected graph containing n vertices represented as a two-dimensional boolean array W[1…n][1…n] , where W[i][j] is true if there is an edge between i and j.

Output: all possible colorings of the graph using at most 2 colors so that no two adjacent vertices are the same color. The output for each coloring is an array vcolor[1…n], where vcolor[i] is the color (1 or 2) assigned to the ith vertex.

```
void twoColor() {
        bool visited[1…n];  //initialized to all false
        Queue q;  //standard queue
        index i;
        for(i=1; i<=n; i++) {
                if(!visited[i]) {
                        insert(q, i);
                        visited[i] = true;
                        vcolor[i] = 1;
                }
                while(q is not empty) {
                        remove(q, current);
                        for(j=current+1; j<=n; j++)
                           if(W[i][j] && !visited[j]) {
                                insert(q, j);
                                visited[j] = true;
                                vcolor[j] = (vcolor[current] % 2) +1; //opposite color
                           }
                        }
                } //end while
                cout >> vcolor[1] through vcolor[n]; //print coloring
                for(i=1; i<=n; i++)
                        vcolor[i] = (vcolor[i] % 2) +1;
                cout >> vcolor[1] through vcolor[n]; //print inverse coloring
    }
```

Complexity is O(n^2).

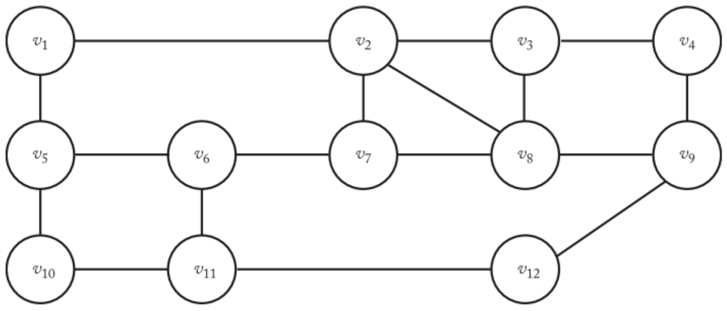
25) Job scheduling with conflicts can be formulated as an m-coloring problem. Jobs must be scheduled into certain time slots, but some jobs conflict with each other (e.g., because they share the same resource). Each node represents a job and each edge represents a conflict between two jobs. A valid coloring represents a valid assignment schedule. The minimum number of colors that can color the graph is the optimal time to finish all of the jobs without conflicts.

Assignment of radio frequencies to radio stations can be formulated as an m-coloring problem. Nodes are radio stations and edges represent conflicts, which are based on the minimum distance between stations that doesn't cause interference. Any two stations closer than this distance have an edge between them. A valid coloring represents a valid assignment of frequencies to radio stations. The minimum number of colors that can color the graph is the minimum number of frequencies needed.

The game Sudoku can be formulated as a 9-coloring of a graph with 81 nodes and as specific set of edges. A valid coloring is a solution to the puzzle. Partial colorings are given as puzzle instances.

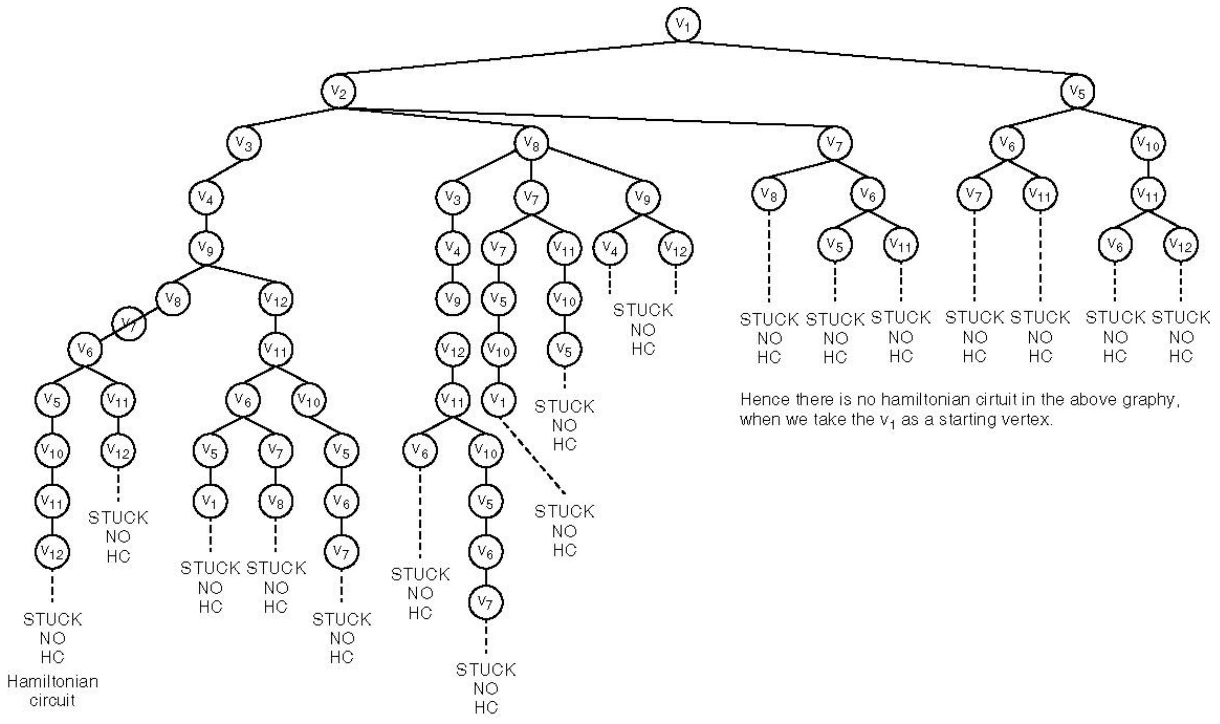
Section 5.6

26) Use the Backtracking algorithm for the Hamiltonian Circuits problem (Algorithm 5.6) to find all possible Hamiltonian Circuits of the following graph.



Show the actions step by step.

Solution:


STUCK NO HC
Hamiltonian circuit
Hence there is no hamiltonian cirtuit in the above graphy, when we take the v1 as a starting vertex.

Hence there is no Hamiltonian circuit in the above graph, because, if there were one, any vertex can be made into the starting vertex.

27) Implement the Backtracking algorithm for the Hamiltonian Circuits problem (Algorithm 5.6) on your system, and run it on the problem instance of Exercise 26.

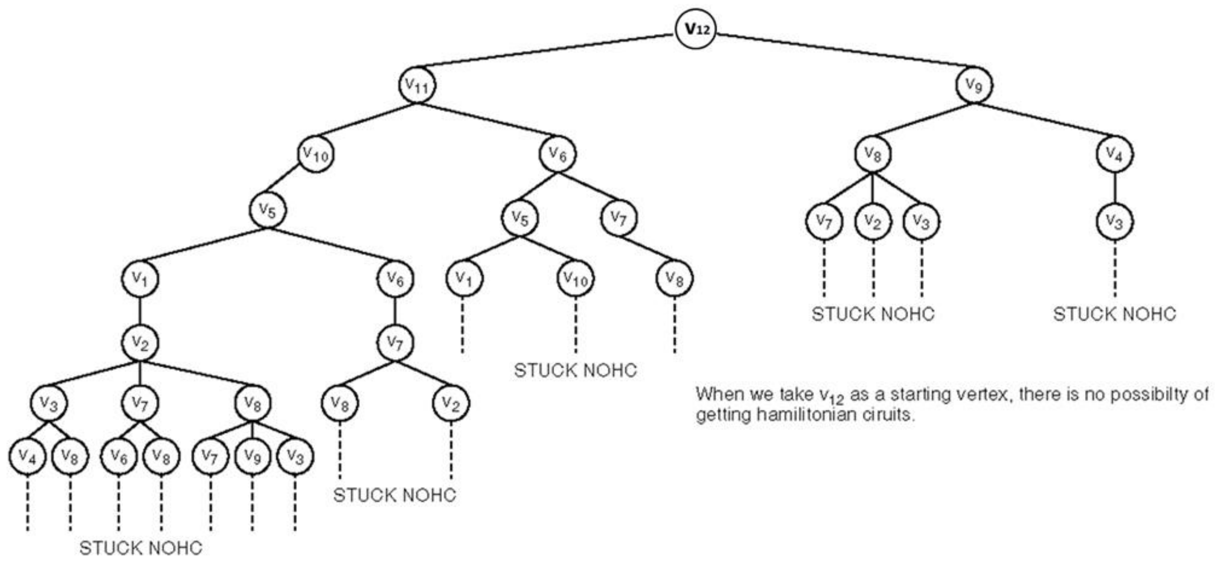
Solution:

```
hamiltonian(index i) {
        index j;
        if (promising (i))
        if (i==n−1)
                    cout << vindex [0] through index [n−1];
        else
                    for (j=1; j<n; j++) {
                              vindex [i+1]=j;
                              hamiltonian(i+1);
                    }
}
```

```
bool promising (index i) {
    index j;
    bool switch;
    if (i==n−1 !w [index [n−1]][vindex [0]])
        switch = false;
    else if (i >0 && !w [vindex[i−1]][vindex[i]])
        switch = false;
    else {
        switch true;
        j=1;
        while (j<i && switch) {
            if (vindex [i] == vindex [j])
                switch = false;
            j++;
        }
        return switch;
    }
}


void main() {
    int i[];
    hamiltonian[0];
}
```

28) Change the starting vertex in the Backtracking algorithm for the Hamiltonian Circuits problem (Algorithm 5.6) in Exercise27 and compare its performance with that of Algorithm 5.6.

Solution: This is the state space tree for the graph given above starting vertex = v_{12}:


STUCK NOHC
STUCK NOHC
STUCK NOHC
STUCK NOHC
STUCK NOHC
When we take v_{12} as a starting vertex, there is no possibilty of getting hamilitonian ciruits.

When we take v_{12} as a starting vertex, there is still no Hamiltonian circuit. This is as expected from Exercise 26, because, if there were one, any vertex could be made into the starting vertex.

Performance comparison: Only 33 nodes searched, compared to 60 at Ex.26.

29) void hamiltonian(index i) {

```
void hamiltonian(index i) {
    index j;
    if(promising(i)) {
        if(i==n-1) {
            cout << vindex[0] through vindex[n-1];
            return;
        }
        for(j=1; j<n; j++) {
            vindex[i+1] = j;
            hamiltonian(i+1);
        }
    }
}
```

Promising function is identical to the one on page 233.

This algorithm has the same worst-case complexity as Algorithm 5.6 since it must check the same number of nodes in the case where there is no Hamiltonian circuit.

30) Analyze the Backtracking algorithm for the Hamiltonian Circuits problem (Algorithm 5.6) and show the worst-case complexity using order notation.

<u>Solution</u>: For the worst case time complexity, the number of nodes in the state space tree searched is given on p.233 as

$$1 + (n-1) + (n-1)^2 + (n-1)^3 + \ldots + (n-1)^n = \frac{(n-1)^{n+1}-1}{n-2}$$

With order notation we have $O(n^{n+1})$.

For a problem with n = 12, we have

$$= \frac{(12-1)^{12+1}-1}{12-2}$$

$$= \frac{(11)^{13}-1}{10}$$

$$= \frac{34522712143931-1}{10} = \frac{34522712143930}{10} = 3{,}452{,}271{,}214{,}393 \text{ nodes in the worst case.}$$

31) Use the Monte Carlo Technique to estimate the efficiency of the Backtracking algorithm for the Hamiltonian Circuits problem (Algorithm 5.6).

<u>Solution</u>: Use Algorithm 5.3 (Monte Carlo for n-Queens) as template, and adapt it to implement the data types and data structures from Algorithm 5.6 (Hamiltonian Circuits). Only the minimal changes (indicated in **boldface**) were made in the code below:

```
int estimate_Hamiltonian ( int n) {
        index i, j;
        index vindex[n];                 //solution is an array of indices in the cycle
        int m, mprod , numnodes ;
        set_of_index prom_children ;

        i = 0;                           //root
        numnodes = 1;
        m = 1;                           //we assume the root is promising
        mprod = 1;
        while (m != 0 && i !=n) {
                mprod = mprod * m;
                numnodes = numnodes + mprod * n ;
                i++;                     //next level in the tree
                m = 0 ;                  //assume no promising children
                prom_children = ∅;       //Initialize with empty set
                for ( j=0; j<n; j++) {
                        vindex[ i ] = j;
                        i f ( promising ( i ) )  {
                                m++;
                                prom_children = prom_children ∪ { j};
                        }
                }
                i f (m != 0) {
                        j = random selection from prom_children ;
                        vindex[ i ] = j ;
                }
        }
        return numnodes ;
}
```

The *promising* function is the same as in Algorithm 5.6.

Section 5.7

32) Compute the remaining values and bounds after visiting node (4, 1) in Example 5.6 (Section 5.7.1).

Solution:

(4, 2): Item 4 was not chosen, so the profit and total weight are the same as in the parent \$70, 7. There are no further items, so the bound is the current profit \$70.

(2, 2): Item 2 was not chosen, so the profit and total weight are the same as in the parent \$40, 2. Bound is 40 + 50 +10 x 2/5 = \$98.

(3, 3): Profit 40 + 50 = 90. Weight 2 + 10 = 12. Since the item was chosen, the bound stays equal to that of the parent \$98.

(4, 3): Profit 40 + 50 + 10 = \$100. Weight 12 + 5 = 17 > 16, so no bound needed.

(4, 4): Item 4 was not chosen, so the profit and total weight are the same as in the parent \$90, 12. There are no further items, so the bound is the current profit \$90.

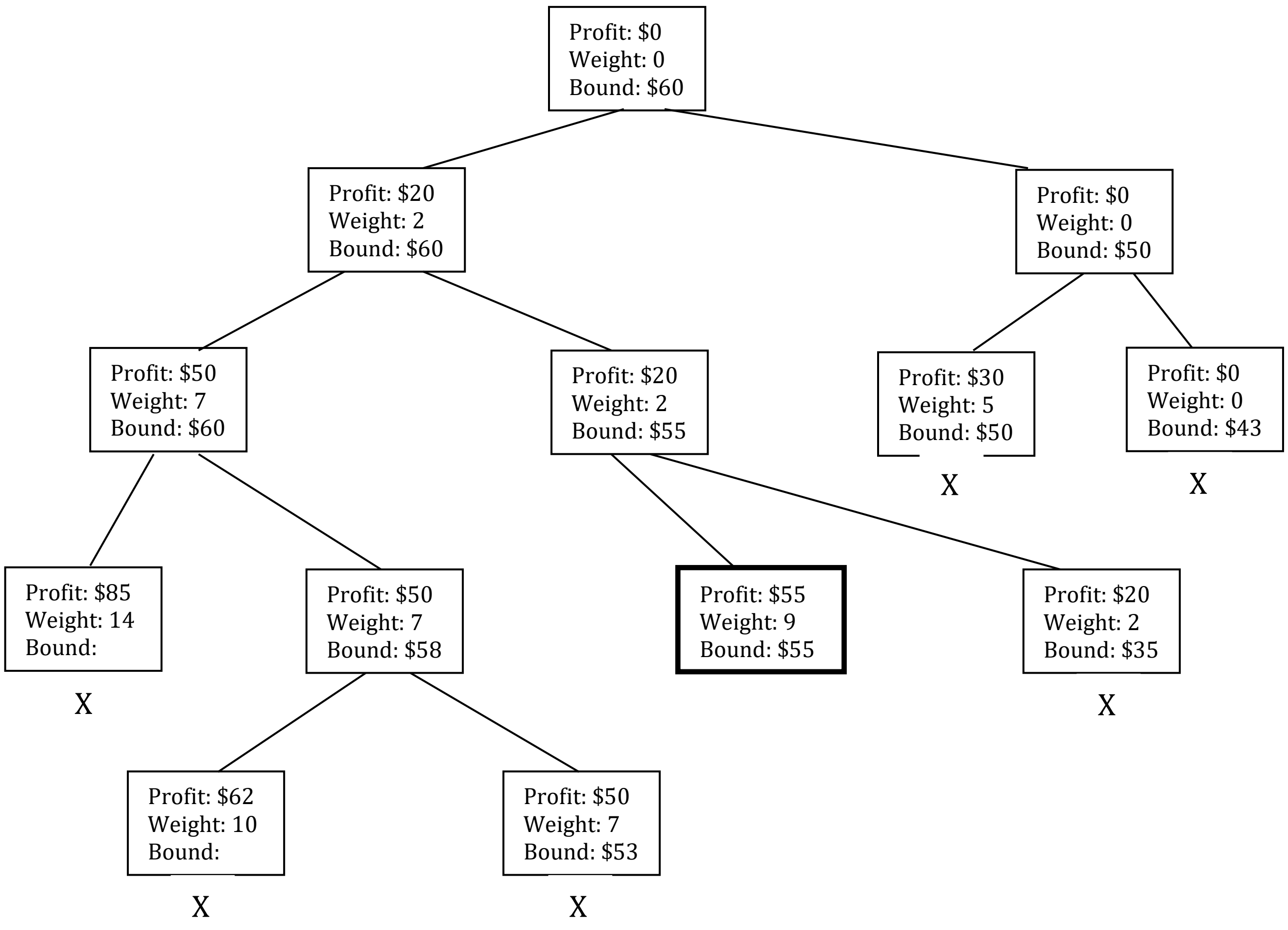
(3, 4): Item 3 was not chosen, so the profit and total weight are the same as in the parent \$40, 2. Bound 40 + 10 = \$50.

(1, 2): Item 1 was not chosen, so the profit and total weight are the same as in the parent \$0, 0. Bound 30 + 50 + 10 x 1/5 = \$82.

33) Use the Backtracking algorithm for the 0-1 Knapsack problem (Algorithm 5.7) to maximize the profit for the following problem instance. Show the actions step by step.

i	p_i	w_i	$\frac{p_i}{w_i}$	
1	\$20	2	10	
2	\$30	5	6	
3	\$35	7	5	$W = 9$
4	\$12	3	4	
5	\$3	1	3	

Solution: *maxprofit* goes through the values \$0, \$20, \$50, \$55. The tree is below:



34) Implement the Backtracking algorithm for the 0-1 Knapsack problem (Algorithm 5.7) on your system, and run it on the problem instance of Exercise 33.

Solutions will vary.

35) Implement the dynamic programming algorithm for the 0-1 Knapsack problem (see Section 4.5.3) and compare the performance of this algorithm with the Backtracking algorithm for the 0-1 Knapsack problem (Algorithm 5.7) using large instances of the problem.

Solutions and performance will vary.

36) Improve the Backtracking algorithm for the 0-1 Knapsack problem (Algorithm 5.7) by calling the promising function after only a move to the right.

<u>Solution</u>: The idea is to pass to the function *knapsack* in Algorithm 5.7 a fourth argument, *eval_promising*, which tells it whether it should evaluate *promising* or not. The changes are shown in **boldface** below:

```
void knapsack (index i, int profit, int weight, bool eval_promising) {
        if (weight <= W && profit > maxprofit) {
                maxprofit = profit;
                numbest = i;
                bestset = include;
        }
        if ( (eval_promising && promising(i)) || !eval_promising ) {
                include [ i + 1] = "yes";          // Include w[ i +1]
                knapsack(i + 1, profit + p[ i + 1] , weight + w[ i + 1], false);
                include [ i + 1] = "no";           // Do not include
                knapsack(i + 1, profit , weight, true); // w[ i + 1].
        }
}
```

The function *promising* is the same as in Algorithm 5.7.

37) Use the Monte Carlo technique to estimate the efficiency of the Backtracking algorithm for the 0-1 Knapsack problem (Algorithm 5.7).

<u>Solution</u>: Use Algorithm 5.3 (Monte Carlo for n-Queens) as template, and adapt it to implement the data types and data structures from Algorithm 5.7. Only the minimal changes (indicated in **boldface**) were made in the code below:

```
int estimate_backpack ( int n) {
    index i ;
    bool j;
    bool include[0…n-1];                //solution is an array of yes/no
    int m, mprod , numnodes , maxprofit;
    set_of_index prom_children ;

    i = 0;   maxprofit = 0;             //root
    numnodes = 1;
    m = 1;                              //we assume the root is promising
    mprod = 1;
    while (m != 0 && i !=n) {
        mprod = mprod * m;
        numnodes = numnodes + mprod * n ;
        i++;                            //next level in the tree
        m = 0 ;                         //assume no promising children
        prom_children = ∅;              //initialize with empty set
        for ( j in {True, False}) {
            include[ i ] = j;
            int maxprof = maxprofit;
            if (j)                      //node i was included
                maxprof = maxprof + p[i];
            i f ( promising ( i ) )  {
                m++;
                prom_children = prom_children ∪ { j};
            }
        }
        i f (m != 0) {
            j = random selection from prom_children ;
            include[ i ] = j ;
        }
    }
    return numnodes ;
}
```

The *promising* function is the same as in Algorithm 5.7.

Additional Exercises

38) 1. Searching for optimal moves in a two-player game (e.g., tic-tac-toe or chess).

2. Discretized path-finding (e.g., optimal routes via waypoints for non-player characters in video games).

3. Anagram solving/generation given an input string and a dictionary.

39) Modify the Backtracking algorithm for the n-Queens problem (Algorithm 5.1) so that it produces only the solutions that are invariant under reflections or rotations.

Solution: The function *queens* of Alg. 5.1, stores each new solution in a global set of solutions, *set_of_solutions*. Before accepting and printing a candidate for a new solution, a check is performed to see if the candidate matches an existing solution through reflection or rotation:

```
set_of_solutions = ∅;        //initialize with empty set

void queens(index i) {
    ... ... ...
    if (promising(i))
         if (i == n)
              if ( !matching_existing(col) ) {
                   set_of_solutions = set_of_solutions U {col};
                   cout << col [1] through col [n];
              }
    ... ... ...
```

The function *matching_existing* has the following pseudocode:

```
bool matching_existing(int col[ ]){
         bool match = false;
         for (sol in set_of_solutions) {
              if   ( match1(col, sol) ) match = true;
              elseif    ( match2(col, sol) ) match = true;
              ... ... ...
              elseif    ( match42(col, sol) ) match = true;
         }
         return match;
    }
```

The functions *match1*, *match2*, … implement checks for the various symmetries that we want to check: horizontal, vertical, with respect to the main diagonal or anti-diagonal, rotation of 90/180/270 deg., etc.

40) Given an n × n × n cube containing n^3 cells, we are to place n queens in the cube so that no two queens challenge each other (so that no two queens are in the same row, column, or diagonal). Can then-Queens algorithm (Algorithm 5.1) be extended to solve this problem? If so, write the algorithm and implement it on your system to solve problem instances in which n = 4 and n = 8.

Solution:

This Algorithm can be extended to solve n × n × n cube n-queens, as described below.

We represent the positions in the cube by a bi-dimensional array of integers *base*[][], which is analogous to *col*[]. We can imagine *base* as being the base of the cube, and the number stored at base[i][j] as being the height of that queen in the cube. The base has n^2 positions, but only n of them are occupied by queens in a solution. To be consistent with the notation in Section 5.2, we use indices 1 through n to store base positions, and integers 1 though n for the height. A height of 0 means no queen in that base position.

In Algorithm 5.1, the condition for two queens being on the same diagonal is

$$col[i] - col[k] = \pm (i-k)$$

Here, the diagonal conditions are

$$base[i1][j1] - base[i2][j2] = \pm (i1 - i2) \text{ AND}$$

$$base[i1][j1] - base[i2][j2] = \pm (j1-j2)$$

The horizontal line condition is

$$base[i1][j1] = base[i2][j2] \text{ AND } (i1 = i2 \text{ OR } j2 = j2)$$

For simplicity, we replace the index type with int, and we define an iterator on the base, which, for any base position (i, j), returns a “next” one: (i1, j1) = next(i, j). The order is arbitrary, as long as, starting from (1, 1), all base positions are reached.

The pseudocode follows closely the one for Algorithm 5.1:

```
void queens3D (int i, int j, int nr_queens) {
        int i1, j1 ;
        if (promising(i, j))
                if (nr_queens == n)
                        print base;
                else
                        for ( k=0; k<=n; k++) {          //k=0 means no queen
                                int new_nr_queens = nr_queens;
                                if (k>0)
                                    new_nr_queens++;
                                (i1, j1) = next(i, j);
                                base[ i1, j1 ] = k ;
                                queens3D(i1, j1, new_nr_queens);
                        }
}
```

For the *promising* function, we assume the existence of a function *before*, which, when called on a pair of indices i, j, returns a set of all the “previous” pairs (according to the function *next* mentioned above).

```
bool promising (int i, int j) {
    int i1, j1;
    bool switch ;

    i1 = 1; j1 = 1;
    switch = true ;
    while ( (i1, j1) is in before(i, j) && switch) {
        if (    ( abs(base[i1][j1] – base[i][j]) == abs(i1 – i2)  &&
                  abs(base[i1][j1] – base[i][j]) == abs(i1 – i2)  )     ||
                ( base[i1][j1] == base[i2][j2] && (i1 == i2)  )         ||
                ( base[i1][j1] == base[i2][j2] && (j2 == j2)  )
        )
                switch = false ;
        (i1, j1) = next(i1, j1);
    }
    return switch ;
}
```

In the main program we call

```
    Queens3D(1, 1, 0);
```

41) Modify the Backtracking algorithm for the Sum-of-Subsets (Algorithm 5.4) to produce the solutions in a variable-length list.

Solution: Instead of using the array *include*, we use a list or set data structure of the same name, which supports the append operation.

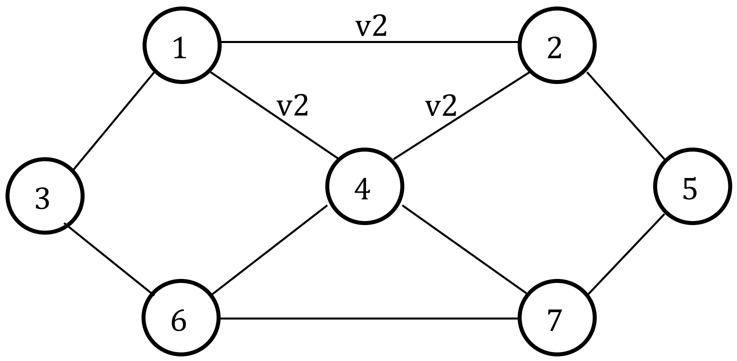
Instead of **include[i+1] = “yes”** we say **include.append(i+1)**

Instead of **include[i+1] = “no”** we do nothing

42) Explain how we can use the Backtracking algorithm for the m-Coloring problem (Algorithm 5.5) to color the edges of the graph of Exercise 18 using the same three colors so that edges with a common end receive different colors.

Solution: For any graph G, we construct the "edge-to-vertex dual graph", a.k.a. the "line graph" G', which has a vertex for every edge in G. Two vertices in G are connected by an edge, if and only if their corresponding edges in are incident on the same node.

For G in exercise 18, the dual G' has 7 vertices, corresponding to the 7 edges of G:



It is easy to see that edges with a common end in G (e.g. the ones incident on v2) correspond to a completely connected subgraph in G' (vertices 1, 2, 4), so those vertices in G' do receive different colors, meaning that the edges in G have different colors.

43) Modify the Backtracking algorithm for the Hamiltonian Circuits problem (Algorithm 5.6) so that it finds a Hamiltonian Circuit with minimum cost for a weighted graph. How does your algorithm perform?

Solution: Hamiltonian circuit problem with weighted graph is also known as the Traveling Salesperson (TSP) problem, first encountered in Section 3.6.

The simplest solutions is to add a function *evaluate* that takes a solution *vindex* as argument and returns its cost, i.e. the sum of the weights of all edges in that solution. We initialize a global variable *minimum* with a value larger than the cost of any cycle (e.g. the sum of all weights in G), and we store the best solution in another global *best_vindex*:

```
minimum = INFINITY;
best_vindex[n] = {-1};

void hamiltonian(index i) {
        index j;
        if(promising(i)) {
                if( i == n – 1) {
                        if (evaluate(vindex) < minimum) {
                                minimum = evaluate(vindex);
                                cout << vindex[0] through vindex[n-1];
                                best_vindex ← vindex;          //use a loop!
                        }
                }
                else
                        for (j=2; j<n; j++) {
                                vindex[i+1] = j;
                                hamiltonian(i+1);
                        }
        }
}
```

The *promising* function is identical to the one on page 233.

To increase efficiency, the *promising* function can also check **evaluate(vindex) < minimum**, so avoids exploration of candidate solutions that are already longer than the current minimum.

44) Modify the Backtracking algorithm for the 0-1Knapsack problem (Algorithm 5.7) to produce a solution in a variable-length list.

Solution: Instead of using the array *include*, we use a list or set data structure of the same name, which supports the append operation.

Instead of **include[i+1] = "yes"** we say **include.append(i+1)**

Instead of **include[i+1] = "no"** we do nothing