Section 6.1

1) Use Algorithm 6.1 (The Breadth-First Search with Branch-and-Bound Pruning algorithm for the 0-1 Knapsack problem) 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
4	\$12	3	4
5	\$3	1	3

$W = 13$

1. Visit node (0,0). Set its profit and weight to \$0 and 0. Compute its bound to be \$80. maxprofit = 0.
2. Compute profits, weights, and bounds for children of (0,0): (1,1) and (1,2).
3. (1,1) has profit of \$20, weight of 2, and bound of \$80. Enqueue node.
4. (1,2) has profit of \$0, weight of 0, and bound of \$69. Enqueue node.
5. Dequeue/visit node (1,1). maxprofit = 20.
6. Compute profits, weights, and bounds for children of (1,1): (2,1) and (2,2).
7. (2,1) has profit of \$50, weight of 7, and bound of \$80. Enqueue node.
8. (2,2) has profit of \$20, weight of 2, and bound of \$70. Enqueue node.
9. Dequeue/visit node (1,2).
10. Compute profits, weights, and bounds for children of (1,2): (2,3) and (2,4).
11. (2,3) has profit of \$30, weight of 5, and bound of \$69. Enqueue node.
12. (2,4) has profit of \$0, weight of 0, and bound of \$50. Enqueue node.
13. Dequeue/visit node (2,1). maxprofit = 50.
14. Compute profits, weights, and bounds for children of (2,1): (3,1) and (3,2).
15. (3,1) has profit of \$85, weight of 14, and bound of \$0. Since weight is exceeded, we do not enqueue.
16. (3,2) has profit of \$50, weight of 7, and bound of \$65. Enqueue node.
17. Dequeue/visit node (2,2).
18. Compute profits, weights, and bounds for children of (2,2): (3,3) and (3,4).
19. (3,3) has profit \$55, weight 9, and bound \$70. Enqueue node.

20. (3,4) has profit $20, weight 2, and bound $35. Since bound is less than maxprofit, we do not enqueue.
21. Dequeue/visit, node (2,3).
22. Compute profits, weights, and bounds for children of (2,3): (3,5) and (3,6).
23. (3,5) has profit $65, weight 12, and bound $69. Enqueue node.
24. (3,6) has profit $30, weight 5, and bound $45. Since bound is less than maxprofit, we do not enqueue.
25. Dequeue/visit node (3,2).
26. Compute profits, weights, and bounds for children of (3,2): (4,1) and (4,2).

23. (4,1) has profit $62, weight 10, and bound $65. Enqueue node.
24. (4,2) has profit $50, weight 7, and bound $53. Enqueue node.
25. Dequeue/visit node (3,3). maxprofit = 55.
26. Compute profits, weights, and bounds for children of (3,3): (4,3) and (4,4).

23. (4,3) has profit $67, weight 12, and bound $70. Enqueue node.
24. (4,4) has profit $55, weight 9, and bound $58. Enqueue node.
25. Dequeue/visit node (3,5). maxprofit = 65.
26. Compute profits, weights, and bounds for children of (3,5): (4,5) and (4,6).

23. (4,5) has profit $77, weight 15, and bound $0. Since weight exceeds limit, we do not enqueue.
24. (4,6) has profit $65, weight 12, and bound $68. Enqueue node.
25. Dequeue/visit node (4,1).
26. Bound for (4,1) is $65, which is less than maxprofit, so we do not check its children.
27. Dequeue/visit node (4,2).
28. Bound for (4,2) is $53, which is less than maxprofit, so we do not check its children.
29. Dequeue/visit node (4,3). maxprofit = 67.
30. Compute profits, weights, and bounds for children of (4,3): (5,1) and (5,2).

23. (5,1) has profit $70, weight 13, and bound $70. Enqueue node.
24. (5,2) has profit $67, weight 12, and bound $67. Since bound does not exceed maxprofit, we do not enqueue.
25. Dequeue/visit node (4,4).
26. Bound for (4,4) is $58, which is less than maxprofit, so we do not check its children.
27. Dequeue/visit node (4,6).
28. Compute profits, weights, and bounds for children of (4,6): (5,3) and (5,4).

23. (5,3) has profit $68, weight 13, and bound $68. Enqueue node.

24. (5,4) has profit $65, weight 12, and bound $65. Since bound does not exceed maxprofit, we do not enqueue.
25. Dequeue/visit node (5,1). maxprofit = 70.
26. Bound for (5,1) is $70, which does not exceed maxprofit, so we do not check its children.
27. Dequeue/visit node (5,3).
28. Bound for (5,3) is $68, which is less than maxprofit, so we do not check its children.
29. Queue is now empty (no promising nodes left) so problem is solved. Solution is set of items given by node (5,1): { 1, 3, 4, 5} which has profit $70.

2) Implement Algorithm 6.1 on your system and run it on the problem instance of Exercise 1.

```
/*Solves 0-1 knapsack with Algorithm 6.1
  (breadth-first search with Branch-and-Bound pruning)
  Tested with MSVS 2012 professional                                  */
#include <iostream>
#include <queue>
#include <list>
using namespace std;
#define W 13
#define n 5
//As in text pseudocode, we start arrays at index 1
int p[] = {0, 20, 30, 35, 12, 3}; //Must be sorted descending by pi/wi!
int w[] = {0,  2,  5,  7,  3, 1};
int node_counter = 0;             //Used to keep track of how many nodes are accessed

struct node{
       int level;
       int profit;
       int weight;
       list<int> items;          //items included on the path to this node
};

void print_list(list<int> x){
       list<int>::iterator i;
       for (i = x.begin(); i != x.end(); i++)
              cout <<*i <<" ";
       cout <<endl;
}

void print_node(struct node u){
       cout <<u.level <<" " <<u.profit <<" " <<u.weight <<" ";
       print_list(u.items);
}

float bound(struct node u){
```

```
	int j, k, totweight;
	float result;
	if (u.weight > W)
		return 0;			//bound is 0 if limit is exceeded
	else{
		result = u.profit;
		j = u.level + 1;	//start adding next items
		totweight = u.weight;
		while (j <= n && totweight + w[j] <= W){
			totweight += w[j];
			result += p[j];
			j++;
		}
		k = j;
		if (k <= n)			//add a fraction of the remaining item
			result += ((float)(W - totweight))*p[k]/w[k];
		return result;
	}
}

int main(){
	struct node u, v;
	queue<struct node> Q;
	list<int> bestitems;	//stores best set of included items so far
	list<int> empty;
	int maxprofit = 0;

	v.level = 0; v.profit = 0; v.weight = 0; v.items = empty; //root has no items
	Q.push(v);
	while (!Q.empty()){
		v = Q.front();	Q.pop();  node_counter++;
		u.level = v.level + 1;
			//Left child includes the next item
		u.weight = v.weight + w[u.level];
		u.profit = v.profit + p[u.level];
		u.items = v.items; u.items.push_back(u.level);
		if (u.weight <= W && u.profit > maxprofit){
			maxprofit = u.profit;
			bestitems = u.items;
		}
		if (bound(u) > maxprofit)
			Q.push(u);
			//Right child does not include the next item
		u.weight = v.weight;
		u.profit = v.profit;
		u.items  = v.items;
		if (bound(u) > maxprofit)
			Q.push(u);
	}
	cout <<"maxprofit = " <<maxprofit <<endl;
	cout <<"list of best items (their indices): ";
	print_list(bestitems);
	cout <<"\n\nnr. of nodes examined (popped) = " <<node_counter <<endl <<endl;
}
```

```
maxprofit = 70
list of best items (their indices): 1 3 4 5


nr. of nodes examined (popped) = 11
```

3) Problem: Let *n* items be given, where each item has a weight and a profit. The weights and profits are positive integers. Furthermore, let a positive integer *W* be given. Determine a set of items with maximum total profit, under the constraint that the sum of their weights cannot exceed *W*.

Inputs: positive integers *n* and *W*, arrays of positive integers *w* and *p,* each indexed from 1 to *n*, and each of which is sorted in nonincreasing order according to the values of *p*[i]/*w*[i].

Outputs: an optimal set of items whose total weight does not exceed *W*.

```
void knapsack2(int n, const int p[], const int w[], int W, set_of_integers& maxset) {
    queue_of_node Q;
    node u, v;
    initialize(Q);
    v.level = 0;    v.profit = 0;    v.weight = 0;  v.items = ∅;
    int maxprofit = 0;
    maxset = ∅;
    enqueue(Q, v);
    while(!empty(Q)) {
        dequeue(Q, v);
        u.level = v.level + 1;
        u.weight = v.weight + w[u.level];
        u.profit = v.profit + p[u.level];
        u.items = v.items;
        add u.level to u.items;
        if(u.weight <= W && u.profit > maxprofit)
            maxprofit = u.profit;
            maxset = u.items;
        }
        if(bound(u) > maxprofit)
            enqueue(Q,u);
        u.weight = v.weight;
        u.profit = v.profit;
        u.items = v.items;
        if(bound > maxprofit)
            enqueue(Q, u);
```

```
        }
    }
```

The bound() function is the identical to the one on page 250.

4) Use Algorithm 6.2 (The Best-First Search with Branch-and-Bound Pruning algorithm for the 0-1 Knapsack problem) to maximize the profit for the problem instance of Exercise 1. Show the actions step by step.

1. Visit node (0,0). Set its profit and weight to $0 and 0. Compute its bound to be $80. maxprofit = 0.
2. Compute profits, weights, and bounds for children of (0,0): (1,1) and (1,2).
3. (1,1) has profit of $20, weight of 2, and bound of $80. Enqueue with key 80.
4. (1,2) has profit of $0, weight of 0, and bound of $69. Enqueue with key 69.
5. Dequeue/visit highest priority node: (1,1). maxprofit = 20.
6. Compute profit, weights, and bounds for children of (1,1): (2,1) and (2,2).
7. (2,1) has profit of $50, weight of 7, and bound of $80. Enqueue with key 80.
8. (2,2) has profit of $20, weight of 2, and bound of $70. Enqueue with key 70.
9. Dequeue/visit highest priority node: (2,1). maxprofit = 50.
10. Compute profit, weights, and bounds for children of (2,1): (3,1) and (3,2).
11. (3,1) has profit of $85, weight of 14, and bound of $0. Since weight is over limit 13, we do not enqueue.
12. (3,2) has profit of $50, weight of 7, and bound of $65. Enqueue with key 65.
13. Dequeue/visit highest priority node: (2,2).
14. Compute profit, weights, and bounds for children of (2,2): (3,3) and (3,3).
15. (3,3) has profit of $55, weight of 9, and bound of $70. Enqueue with key 70.
16. (3,4) has profit of $20, weight of 2, and bound of $35. Since bound is less than maxprofit, we do not enqueue.
17. Dequeue/visit highest priority node: (3,3). maxprofit = $55.
18. Compute profit, weights, and bounds for children of (3,3): (4,1) and (4,2).
19. (4,1) has profit of $67, weight of 12, and bound of $70. Enqueue with key 70.
20. (4,2) has profit of $55, weight of 9, and bound of $58. Enqueue with key 58.
21. Dequeue/visit highest priority node: (4,1). maxprofit = $67.
22. Compute profit, weights, and bounds for children of (4,1): (5,1) and (5,2).
23. (5,1) has profit of $70, weight of 13, and bound of $70. Enqueue with key 70.

24. (5,2) has profit of $67, weight of 12, and bound of $67. Since bound does not exceed maxprofit, we do not enqueue.
25. Dequeue/visit highest priority node: (5,1). maxprofit = $70.
26. Since bounds of all remaining nodes are less than or equal to maxprofit, problem is solved. Solution is set of items given by node (5,1): { 1, 3, 4, 5} which has profit $70.

5) Implement Algorithm 6.2 on your system and run it on the problem instance of Exercise 1.

```
/*Solves 0-1 knapsack with Algorithm 6.2
  (best-first search with Branch-and-Bound pruning)
  Tested with MSVS 2012 professional                                    */
#include <iostream>
//#include <queue>
#include <list>
using namespace std;

#define W 13
#define n  5
int p[]      = {0, 20, 30, 35, 12, 3}; //Must be sorted descending by pi/wi!
int w[]      = {0,  2,  5,  7,  3, 1};
int node_counter = 0;     //Used to keep track of how many nodes are accessed

struct node{
       int level;
       int profit;
       int weight;
       float boundie;
       list<int> items;          //items included on the path to this node
};

void print_list(list<int> x){
       list<int>::iterator i;
       for (i = x.begin(); i != x.end(); i++)
              cout <<*i <<" ";
       cout <<endl;
}

void print_node(struct node u){
       cout <<u.level <<" " <<u.profit <<" " <<u.weight <<" ";
       print_list(u.items);
}

bool compare_nodes(const struct node a, const struct node b){
       if (a.boundie < b.boundie)
              return true;
       else
              return false;
}

float bound(struct node u){
       int j, k, totweight;
```

```
        float result;
        if (u.weight > W)
                return 0;                   //bound is 0 if limit is exceeded
        else{
                result = u.profit;
                j = u.level + 1;    //start adding next items
                totweight = u.weight;
                while (j <= n && totweight + w[j] <= W){
                        totweight += w[j];
                        result += p[j];
                        j++;
                }
                k = j;
                if (k <= n)                 //add a fraction of the remaining item
                        result += ((float)(W - totweight))*p[k]/w[k];
                return result;
        }
}

int main(){
        struct node u, v;
        list<struct node> PQ;      //Priority Queue implemented with std::list
                                            //push at front, pop at back
        list<int> bestitems;       //stores best set of included items so far
        list<int> empty;
        int maxprofit = 0;

        v.level = 0; v.profit = 0; v.weight = 0; v.boundie = bound(v);
        v.items = empty; //root has no items
        PQ.push_front(v);
        while (!PQ.empty()){
                v = PQ.back();      PQ.pop_back();  node_counter++;
                if (v.boundie > maxprofit){
                        u.level  = v.level + 1;
                                    //Left child includes the next item
                        u.weight  = v.weight + w[u.level];
                        u.profit  = v.profit + p[u.level];
                        u.boundie = bound(u);
                        u.items = v.items; u.items.push_back(u.level);
                        if (u.weight <= W && u.profit > maxprofit){
                                maxprofit = u.profit;
                                bestitems = u.items;
                        }
                        if (u.boundie > maxprofit){
                                PQ.push_front(u);   //push_back also works, since we're sorting it
                                PQ.sort(compare_nodes);    //keep PQ sorted
                        }
                                    //Right child does not include the next item
                        u.weight = v.weight;
                        u.profit = v.profit;
                        u.items  = v.items;
                        u.boundie = bound(u);
                        if (u.boundie > maxprofit){
                                PQ.push_front(u);   //push_back also works, since we're sorting it
                                PQ.sort(compare_nodes);    //keep PQ sorted
                        }
```

```
            }
        }
        cout <<"Knapsack with BEST-FIRST Branch-and-bound" <<endl <<endl;
        cout <<"maxprofit = " <<maxprofit <<endl;
        cout <<"list of best items (their indices): ";
        print_list(bestitems);
        cout <<"nr. of nodes examined (popped)    : " <<node_counter <<endl <<endl;
}
```

```
Knapsack with BEST-FIRST Branch-and-bound

maxprofit = 70
list of best items (their indices): 1 3 4 5
nr. of nodes examined (popped)    : 8
```

Note that, even for this small problem, the Best-First algorithm is better than the Breadth-First: it found the solution after examining only 8 nodes instead of the 15 from Exercise 2.

6) Compare the performance of Algorithm 6.1 with that of Algorithm 6.2 for large instances of the problem.

Below are the solutions for this Knapsack problem with 15 items (already sorted descending by pi/wi)

```
#define W 750
#define n 15
int p[] =
{0,240,192,184,229,149,135,173,221,210,139,156,201,214,150,163};
int w[] =
{0,120, 98, 94,118, 77, 70, 90,115,110, 73, 82,106,113, 80, 87};
```

```
Knapsack with BREADTH-FIRST Branch-and-bound

maxprofit = 1458
list of best items (their indices): 1 2 3 4 5 6 7 11
nr. of nodes examined (popped)    : 909
```

```
Knapsack with BEST-FIRST Branch-and-bound

maxprofit = 1458
list of best items (their indices): 1 2 3 4 5 6 7 11
nr. of nodes examined (popped)    : 106
```

The Best-First algorithm is much better than Breadth-First: it finds the solution after examining only 106 nodes instead of 909.

Section 6.2

7) 1. Visit node [1]. Compute bound to be 28. minlength = ∞.

2. Visit node [1,2]. Compute bound to be 28.

3. Visit node [1,3]. Compute bound to be 31.

4. Determine next most promising node: [1,2].

5. Visit node [1,2,3]. Compute bound to be 28.

6. Visit node [1,2,5]. Compute bound to be ∞ (there are no edges from v_5 to vertices not already in the path).

7. Determine next most promising node:: [1,2,3].

8. Visit node [1,2,3,4]. Compute bound to be 28.

9. Visit node [1,2,3,7]. Compute bound to be 31.

10. Determine next most promising node:: [1,2,3,4].

11. Visit node [1,2,3,4,8]. Compute bound to be 28.

12. Determine next most promising node: [1,2,3,4,8].

13. Visit node [1,2,3,4,8,7]. Compute bound to be 28.

14. Visit node [1,2,3,4,8,6]. Compute bound to be 29.

15. Determine next most promising node: [1,2,3,4,8,7].

16. Visit node [1,2,3,4,8,7,6], which completes tour [1,2,3,4,8,7,6,5,1] with length 33. Set minlength to 33.

17. Determine next most promising node: [1,2,3,4,8,6].

18. Visit node [1,2,3,4,8,6,5]. Compute bound to be ∞ (there are no edges from v_6 to vertices not already in the path).

19. Determine next most promising node: [1,2,3,7] (ties broken arbitrarily).

20. Visit node [1,2,3,7,4]. Compute bound to be 31.

21. Visit node [1,2,3,7,6]. Compute bound to be 36. Since bound is greater than minlength, this node is not promising.

22. Determine next most promising node: [1,3] (ties broken arbitrarily).

23. Visit node [1,3,4]. Compute bound to be 31.

24. Visit node [1,3,7]. Compute bound to be 34. Since bound is greater than minlength, this node is not promising.

25. Determine next most promising node: [1,3,4] (ties broken arbitrarily).

26. Visit node [1,3,4,8]. Compute bound to be 31.

27. Determine next most promising node: [1,3,4,8] (ties broken arbitrarily).

28. Visit node [1,3,4,8,7]. Compute bound to be 31.

29. Visit node [1,3,4,8,6]. Compute bound to be 32.

30. Determine next most promising node: [1,3,4,8,7].

31. Visit node [1,3,4,8,7,6]. Compute bound to be 36. Since bound is greater than minlength, this node is not promising.

32. Determine next most promising node: [1,2,3,7,4].

33. Visit node [1,2,3,7,4,8]. Compute bound to be 31.

34. Determine next most promising node: [1,2,3,7,4,8].

35. Visit node [1,2,3,7,4,8,6], which completes tour [1,2,3,7,4,8,6,5,1] with length 32. Set minlength to 32.

36. Node [1,3,4,8,6] becomes not promising.

37. There are no more promising nodes, so problem is solved. Solution is given by node [1,2,3,7,4,8,6,5,1], which has cost 32.

8) 1. Visit node [1]. Compute bound to be 27. minlength = ∞.

2. Visit node [1,2]. Compute bound to be 27.

3. Visit node [1,3]. Compute bound to be 27.

4. Visit node [1,4]. Compute bound to be 31.

5. Visit node [1,5]. Compute bound to be 29.

6. Determine next most promising node: [1,2] (tie broken arbitrarily).

7. Visit node [1,2,3]. Compute bound to be 36.

8. Visit node [1,2,4]. Compute bound to be 31.

9. Visit node [1,2,5]. Compute bound to be 30.

10. Determine next most promising node: [1,3].

11. Visit node [1,3,2]. Compute bound to be 27.

12. Visit node [1,3,4]. Compute bound to be 34.

13. Visit node [1,3,5]. Compute bound to be 40.

14. Determine next most promising node: [1,3,2].

15. Visit node [1,3,2,4], which completes tour [1,3,2,4,5,1] with length 31. Set minlength to 31. Nodes [1,4], [1,2,3], [1,2,4], [1,3,4] and [1,3,5] are no longer promising.

16. Visit node [1,3,2,5], which completes tour [1,3,2,5,4,1] with length 30. Set minlength to 30. Node [1,2,5] is no longer promising.

17. Determine next most promising node: [1,5].

18. Visit node [1,5,2]. Compute bound to be 31. This node is not promising.

19. Visit node [1,5,3]. Compute bound to be 33. This node is not promising.

20. Visit node [1,5,4]. Compute bound to be 29.

21. Determine next most promising node: [1,5,4].

22. Visit node [1,5,4,2], which completes tour [1,5,4,2,3,1] with cost 46.

23. Visit node [1,5,4,3], which completes tour [1,5,4,3,2,1] with cost 33.

24. There are no more promising nodes, so problem is solved. Minimum length tour is given by node [1,3,2,5,4,1] and has cost 30.

9) Write functions length and bound used in Algorithm 6.3.

Both functions assume the ordered set given in u.path has a function next() that, starting with the first element of the set, gives the next element in the set, and a size() function, which gives the number of elements in the set.

```
int length(node u) {
    int sum = 0;
    index i;
    index current, next = u.path.next();
    for(i=1; i<u.path.size(); i++) {
        current = next;
```

```
            next = u.path.next();
            sum += W[current][next];
        }
        return sum;
    }

    int bound(node u) {
        index i, j;
        index sum = 0;
        index first = u.path.next();
        index current, next = first;
        //compute current path cost
        for(i=1; i<=u.path.size(); i++) {
            current = next;
            next = u.path.next();
            sum += W[current][next];
        }
        //compute minimum of remaining length
        int min = ∞;
        //compute min weight out of last vertex in path
        for(all j such that 2 <= j <= n && j is not in u.path) {
            if(W[next][j] < min)
                min = W[next][j];
        sum += min;
        //compute min weight into first vertex in path
        for(all j such that 2 <= j <= n && j is not in u.path) {
            if(W[j][first] < min)
                min = W[next][j];
        sum += min;
        //compute min weight out of all remaining vertices
        for(all i such that 2 <= i <= n && i is not in u.path) {
            min = ∞;
            for(all j such that 2 <= j <= n && j is not in u.path)
```

```
                    if(W[i][j] < min)
                        min = W[i][j];
                sum += min;
        }
        return sum;
    }
```

10) Consider the Traveling Salesperson problem.

(a) Write the brute-force algorithm for this problem that considers all possible tours.

Solution: This was solved using backtracking in Exercise 43 of Ch.5:

```
minimum = INFINITY;
best_vindex[n] = {-1};
void hamiltonian(index i) {
      index j;
      if(promising(i)) {
            if( i == n - 1  && evaluate(vindex) < minimum) {
                  minimum = evaluate(vindex);
                  cout << vindex[0] through vindex[n-1];
            }
            for(j=2; j<n; j++) {
                  vindex[i+1] = j;
                  hamiltonian(i+1);
            }
      }
  }
```

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

The function *evaluate* takes a solution *vindex* as argument and returns its cost, i.e. the sum of the weights of all edges in that solution.

(b) Implement the algorithm and use it to solve instances of size 6, 7, 8, 9, 10, 15, and 20.

Solution:

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

```
#define n 9
int W[n+1][n+1] = { {0,  0,  0,  0,  0,  0,  0,  0,  0,  0},
                    {0,  0,  7, 37, 34, 13, 48, 50, 26, 45},
                    {0, 36,  0,  2, 21, 16,  3, 17,  2,  5},
                    {0, 14, 14,  0, 44, 10, 29, 22,  2,  6},
                    {0, 22,  6,  5,  0,  4, 11, 15,  8, 43},
                    {0, 17, 14, 27, 28,  0, 22, 20,  5, 21},
                    {0, 38, 33, 34, 41, 49,  0, 33, 27, 32},
                    {0, 19, 15, 45, 37, 34, 28,  0,  2, 25},
                    {0, 40, 28, 36, 18, 45, 23, 27,  0, 28},
                    {0, 17, 48, 37,  5,  4, 34, 36, 12,  0}  };
int best_vindex[n] = {-1};
int vindex[n] = {-1};
int minimum = 1000000;

bool promising(int i){
      int j;
      bool switchie;
      if (i==n-1 && !W[vindex[n-1]][vindex[0]])     //No edge to close circuit
            switchie = false;
      else if (i>0 && !W[vindex[i-1]][vindex[i]])   //No edge to current node
            switchie = false;
      else{                              //Check if vertex has already been selected
            switchie = true;
            j = 1;
            while (j<i && switchie){
                  if (vindex[i] == vindex[j])
                        switchie = false;
                  j++;
            }
      }
      return switchie;
}

int evaluate(int vindex[]){
      int sum = 0;
      for (int i=0; i<n-1 ;i++)
            sum += W[vindex[i]][vindex[i+1]];
      sum += W[vindex[n-1]][vindex[0]];             //closing the circuit
      return sum;
}

void hamiltonian(int i) {
      int j;
      if(promising(i)) {
            if (i == n-1){
                  int eva = evaluate(vindex);
                  if (eva < minimum) {
                        minimum = eva;
                        cout <<"Found new minimum tour = " <<minimum <<" : ";
                        for (int k=0; k<n; k++){
                              cout << vindex[k] <<" ";
                              best_vindex[k] = vindex[k]; //update minimal circuit
                        }
                        cout <<endl;
```

```
                }
            }
            else
                for(j=2; j<=n; j++) {
                    vindex[i+1] = j;
                    hamiltonian(i+1);
                }
        }
}

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

        vindex[0] = 1;
        hamiltonian(0);

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

Below are the minimal tours obtained for problems with n = 9, 10, 12, and 15 cities:

```
Found new minimum tour = 104 : 1 2 3 9 4 5 8 6 7
Found new minimum tour = 95 : 1 2 6 7 8 4 3 9 5
        time = 0 s
```

```
Found new minimum tour = 90 : 1 2 6 7 8 10 4 3 9 5
Found new minimum tour = 89 : 1 2 6 10 7 8 4 3 9 5
        time = 1 s
```

```
Found new minimum tour = 81 : 1 2 6 12 11 4 3 9 5 8 10 7
Found new minimum tour = 78 : 1 10 7 8 11 4 3 9 5 2 6 12
        time = 64 s
```

```
Found new minimum tour = 321 : 1 11 4 6 14 12 10 8 3 7 5 15 9 2 13
Found new minimum tour = 317 : 1 13 2 15 9 5 7 3 10 14 12 8 6 4 11
        time = 2362 s
```

For n = 9 (and below), the runtime is under 1 second. For n = 15, it is 2,362 seconds. n =20 was not attempted, due to excessive runtime.

(c) Compare the performance of this algorithm to that of Algorithm 6.3 using the instances developed in (b).

Solution: Performances will vary.

11) Implement Algorithm 6.3 on your system, and run it on the problem instance of Exercise 7. Use different bounding functions and study the results.

12) Compare the performance of your dynamic programming algorithm (see Section 3.6, Exercise 27) for the Traveling Salesperson problem with that of Algorithm 6.3 using large instances of the problem.

Performances will vary.

Section 6.3

13) Problem: Determine the *m* most probably sets of diseases (explanations) given a set of *n* diseases and a set *S* of symptoms. It is assumed that if a set of diseases D is a subset of diseases D', then p(D') <= p(D).

Inputs: positive integer *n*, positive integer *m*, a Bayesian network *BN* representing the probabilistic relationships among *n* diseases and their symptoms, and set of symptoms *S*.

Outputs: a priority queue *best* containing *m* sets of indices of diseases representing the *m* most probable sets of diseases, with lower probability diseases having higher priority. The keys of *best* represent the probabilities of these sets, and the data elements in *best* are the sets themselves.

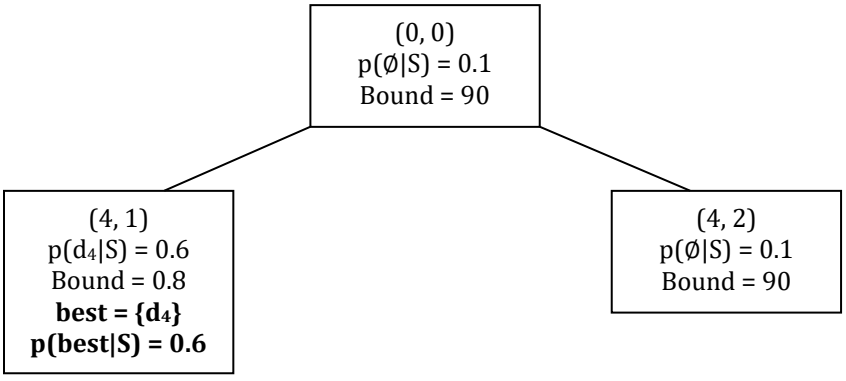
```
void cooper2(int n, int m, Bayesian_network_of_n_diseases BN, set_of_symptoms S,
priority_queue_of_sets_of_indices best) {
    priority_queue_of_node PQ;
    node u, v;
    v.level = 0;
    v.D = ∅;
    insert(best, p(∅|S);
    v.bound = bound(v);
    insert(PQ, v);
    while(!empty(PQ)) {
            remove(PQ);
            if(v.bound > best.peek().key()) {
                    u.level = v.level + 1;
                    u.D = v.D;
                    put u.level in u.D;
```

```
                    if(p(u.D|S) > best.peek().key()) {
                        if(size(best)==m)
                            remove(best);
                        insert(best, u.D, p(u.D|S));
                    }
                    u.bound = bound(u);
                    if(u.bound > best.peek().key())
                        insert(PQ, u);
                    u.D = v.D;
                    u.bound = bound(u);
                    if(u.bound > best.peek().key())
                        insert(PQ, u);
                }
            }
        }
```

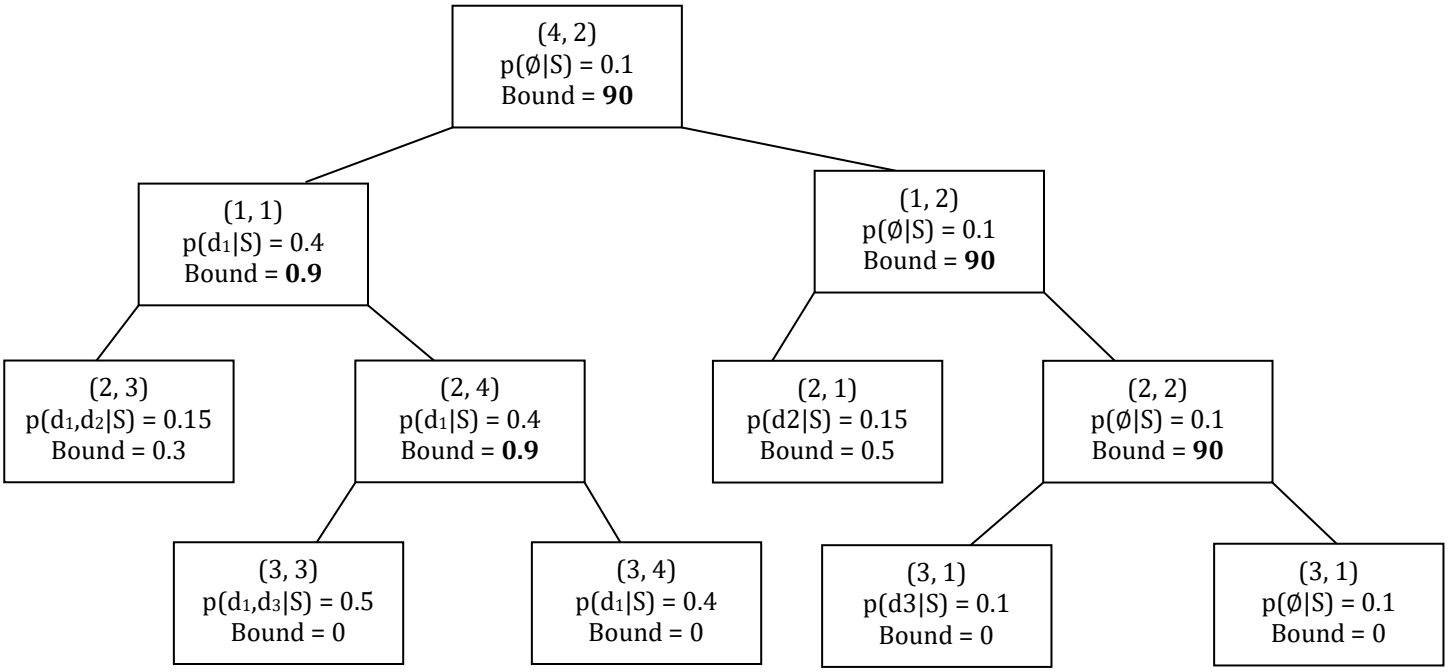
It is assumed that the priority queue *best* is a min-queue (the minimum-valued key is at the head of the queue) and that the function peek() is available to look at the head of the queue without removing it. Also, the key value of the element at the head of the queue can be obtained by calling peek().key(). The insert function for this queue takes in the element to be enqueued as well as the key with which it is inserted (the conditional probability) to be explicit. Finally, the number of elements in the queue can be obtained by calling the size() function. The bound function is identical to the one on page 274.

14) Show that if the diseases in Example 6.4 were sorted in nonincreasing order according to their conditional probabilities, the number of nodes checked would be 23 instead of 15. Assume that p(d4)= 0.008 and p(d4, d1)= 0.007.

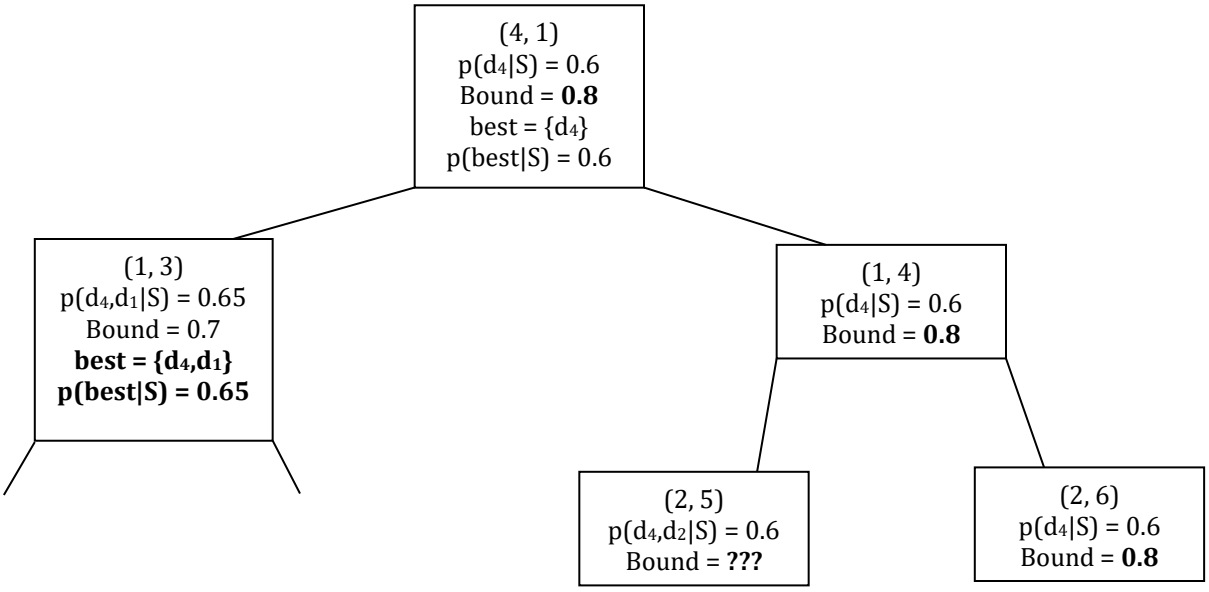
The sorted diseases are: p(d4|S) = 0.6, p(d1|S) = 0.4, p(d2|S) = 0.15, p(d3|S) = 0.1, so we try them in the order d4, d1, d2, d3.



The solution continues with the subtree rooted at (4, 2), b/c the largest bound in a leaf is 90:



At this point, the largest bound in a leaf is in node (4, 1), so we start exploring that subtree:



Not finished. Waiting for feedback from text author.

15) A set of explanations satisfies a comfort measure p if the sum of the probabilities of the explanations is greater than or equal to p. Revise Algorithm 6.4 to produce a set of explanations that satisfies p, where $0 \leq p \leq 1$. Do this with as few explanations as possible.

If the goal is to find the solution with minimum number of diseases, the priority queue should use the nodes' number of diseases to assign priorities (rather than the bound). Higher priority is given to fewer diseases. This is made easier by simply storing the number of diseases in the node structure when the node is created:

struct node{

int level ; // the node's level in the tree

set of indices D;

int nr_indices;

float bound;

}

Since this is not an optimization problem anymore, *best* and *pbest* are not needed.

We still use bound to cull off unpromising nodes by comparing it to the level of comfort p.

```
void comfort (  int n, float p,
                Bayesian network of n diseases BN,
                set of symptoms S  ){
     priority queue of node PQ;
     node u, v;
     v.level = 0;     v.D = ∅;   v.nr_indices = 0;
     v.bound = bound(v);
     insert(PQ, v);
     while (!empty(PQ)){
          remove(PQ, v);         // Remove node with lowest nr_indices.
          if (p(u.D|S) > p){     //Level of comfort reached!
               print D;
               return;
          }
          else {
               u.level = v.level +1; // Set u to a child of v.
```

```
                u.D = v.D; // Set u to the child that
                put u.level in u.D; // includes the next disease .
                u.nr_indices = v.nr_indices + 1; //one more disease
                u.bound = bound(u);
                if (u.bound > p) //node could reach level of comfort
                      insert(PQ, u);
                u.D = v.D;            // Set u to the child that does
                u.bound = bound(u);   // not include the next disease
                u.nr_indices = v.nr_indices;
                if (u.bound > p) //node could reach level of comfort
                      insert(PQ, u);
          }
      }//end while
}
```

The function *bound* is the same as the one in Algorithm 6.4

16) Implement Algorithm 6.4 on your system. The user should be able to enter an integer m, as described in Exercise 11, or a comfort measure p, as described in Exercise 13.

Waiting for feedback from text author.

17) Can the branch-and-bound design strategy be used to solve the problem discussed in Exercise 38 in Chapter 3? Justify your answer.

The problem is: Find the maximum sum in any contiguous sublist in a list of n numbers.

Since the contiguous sublist is a sequence of objects chosen from a specified set (the n numbers), it can be solved through backtracking (brute-force solution): We explore the solution space with a tree similar to any of the ones used in this chapter: the left child designates the solution in which the number was chosen, and the right child – the solution in which it was not.

For branch-and bound, we need a bounding function. Since this is a maximum problem, we need an upper bound on the current candidate solution: we add up all the consecutive non-negative numbers following the current one. We also store in the node the index of the last non-negative number that was added, in order to be able to recover the actual subsequence.

18) Problem: Determine the schedule with the maximum total profit given that each job has a profit that will be obtained only if the job is scheduled by its deadline.

Inputs: positive integer *n* (the number of jobs), an array of integers *deadline* indexed from 1 to *n*, where *deadline*[i] is the deadline for job i, and an array of integers profits, where *profit*[i] is the profit for finishing job i by its deadline. Both deadline and profits have been sorted in nonincreasing order according to the profits of the jobs.

Outputs: an optimal sequence *J* for the jobs.

```
void schedule2(int n, const int deadline[], const int profit[], sequence_of_integer& J) {
    priority_queue_of_node PQ;
    node u, v;
    initialize(PQ);
    v.profit = 0;
    v.sequence = [];
    v.bound = bound(v.sequence);
    insert(PQ, v);
    maxprofit = 0;
    while(!empty(PQ)) {
            remove(PQ, v);
            if(v.bound > maxprofit) {
                    index i;
                    for(each i not in v.sequence) {
                            K = J with i added;
                            if(K is feasible) {
                                    if(profit(K) > maxprofit)
                                            maxprofit = profit(K);
                                    u.sequence = K;
                                    u.bound(u.sequence);
                                    if(u.bound > maxprofit)
                                            insert(PQ, u);
                            }
                    }
            }
    }
```

```
}

int profit(const sequence_of_integer& J) {
    int sum = 0;
    for(each element x in K)
            sum +=  profit[x];
    return sum;
}

int bound(const sequence_of_integer& J) {
    int sum = profit(J);
    sequence_of_integer K = J;
    index i;
    for(i=1; i<=n; i++) {
            if(K does not contain i) {
                    add i to K;
                    sum += profit[i];
            }
    }
    return sum;
}
```

19) Can the branch-and-bound design strategy be used to solve the problem discussed in Exercise 26 in Chapter 4? Justify your answer.

Waiting for feedback from text author.

20) Can the branch-and-bound design strategy be used to solve the Chained Matrix Multiplication problem discussed in Section 3.4? Justify your answer.

Branch-and-bound is only applicable to backtracking algorithms, which, in turn, are only applicable to problems in which a sequence of objects is chosen from a specified set (see Section 5.1). The Chained Matrix Multiplication problem is not described by a sequence, but rather by a binary tree (each pair of parentheses is a node in the tree). Note that in the CMM problem we're not choosing the matrices themselves, but rather the pairs of matrices to multiply. Neither backtracking, nor, consequently, branch-and-bound are applicable.

21) 1. Game-tree search in video games for opponent AI or path-planning. For example, picking the highest scoring move in a chess game according to some heuristic scoring mechanism.

2. Nearest neighbor search in high-dimensional spaces. For example, a kd-tree iteratively splits a set of data points in Euclidian space into a tree of sub-regions, with each region having about half of the number of data points is its parent region. Search can then be performed quickly by checking which region the point would map at each branch of the tree.

3. Generalized integer programming problems (of which the 0-1 Knapsack problem is a specific instance), in which a function must be maximized (or minimized) according to some set of linear constraints, but the variables of the function are constrained to take on integer values.