Section 3.1

1) Establish Equality 3.1 given in this section.

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & k = 0 \text{ or } k = n. \end{cases} \qquad (3.1)$$

If k = 0, the definition of the binomial coefficient

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!} \qquad \text{for } 0 \le k \le n.$$

gives n!/(1·n!) = 1. If k=n, the same definition yields n!/(n!·1) = 1.

We are left with the case 0 < k < n: $\binom{n-1}{k-1} + \binom{n-1}{k} = \frac{(n-1)!}{(k-1)!(n-k)!} + \frac{(n-1)!}{k!(n-k-1)!} =$

$k\frac{(n-1)!}{k!(n-k)!} + (n-k)\frac{(n-1)!}{k!(n-k)!} = (k+n-k)\frac{(n-1)!}{k!(n-k)!} = \frac{n(n-1)!}{k!(n-k)!} = \frac{n!}{k!(n-k)!} = \binom{n}{k}$

2) Use induction on n to show that the divide-and-conquer algorithm for the Binomial Coefficient problem (Algorithm 3.1), based on Equality 3.1, computes

$2\binom{n}{k} - 1$ terms to determine $\binom{n}{k}$.

Induction base: If n = 1, there are two possibilities for 0 ≤ k ≤ n:

- If k == 0, Algorithm 3.1 returns the term directly, so it calculates 1 term. The formula $2\binom{n}{k} - 1$ also gives 2·1 – 1 = 1.
- If k = 1 we have k == n, and Algorithm 3.1 again returns the term directly, so it calculates 1 term. The formula $2\binom{n}{k} - 1$ also gives 2·n – 1 = 2 – 1 = 1.

Induction hypothesis: The result holds up to n.
Induction step: We prove that the result holds for n + 1: The cases k == 0 and k == n are easy, so consider general k: Algorithm 3.1 returns the sum bin(n+1-1, k-1) + bin(n+1-1, k) = bin(n,

k-1) + bin(n,k), which, according to the hypothesis, requires $2\binom{n}{k-1} - 1 + 2\binom{n}{k} - 1 + 1$ (the last 1 counts the addition)= $2(\binom{n}{k-1} + \binom{n}{k}) - 1$. According to formula 3.1, the sum of binomials is $\binom{n+1}{k}$, so we have $2\binom{n+1}{k} - 1$. Q.e.d.

3) Implement both algorithms for the Binomial Coefficient problem (Algorithms 3.1 and 3.2) on your system and study their performances using different problem instances.

```
/*Program that times Binomial function calls*/
#include <time.h>
#include <iostream>
using namespace std;

//Recursive binomial
int bino_rec(int n, int k){
   if ( k == 0 || k == n )/* base case */
      return 1;
   else                                          /* recursive step */
      return bino_rec(n-1, k-1) + bino_rec(n-1, k);
}

int B[101][101];          //for algorithm simplicity only!

//Dynamic programming binomial
int bino_dyn(int n, int k){
       int min;

       for (int i=0; i<=n; i++){
              if (i<k) min = i;
              else min = k;
              for(int j=0; j<=min; j++)
                     if (j==0 || j==i)
                            B[i][j] = 1;
                     else
                            B[i][j] = B[i-1][j-1] + B[i-1][j];
       }
       return B[n][k];
}

int main(){
       time_t begin, end;
       int N, k;
       cout << ("Enter positive integer N(<=100): ");
       cin  >> N;
```

```
        cout << ("Enter positive integer k(<=100): ");
        cin  >> k;
        begin = time(NULL);
        cout << "Binomial recursive     : " << bino_rec(N,k) << endl;
        end    = time(NULL);
        cout << "\ttime = " << difftime(end, begin) << " s" << endl;

        begin = time(NULL);
        cout << "Fibonacci dynamic prog.: " << bino_dyn(N,k) << endl;
        end    = time(NULL);
        cout << "\ttime = " << difftime(end, begin) << " s" << endl <<endl;

        return 0;
}
```

```
Enter positive integer N(<=100): 50
Enter positive integer k(<=100): 5
Binomial recursive       : 2118760
        time = 0 s
Binomial dynamic prog.   : 2118760
        time = 0 s
```

```
Enter positive integer N(<=100): 50
Enter positive integer k(<=100): 6
Binomial recursive       : 15890700
        time = 1 s
Binomial dynamic prog.   : 15890700
        time = 0 s
```

```
Enter positive integer N(<=100): 50
Enter positive integer k(<=100): 7
Binomial recursive       : 99884400
        time = 5 s
Binomial dynamic prog.   : 99884400
        time = 0 s
```

```
Enter positive integer N(<=100): 50
Enter positive integer k(<=100): 8
Binomial recursive       : 536878650
        time = 26 s
Binomial dynamic prog.   : 536878650
        time = 0 s
```

As expected, increasing k makes the recursive algorithm "go exponential".

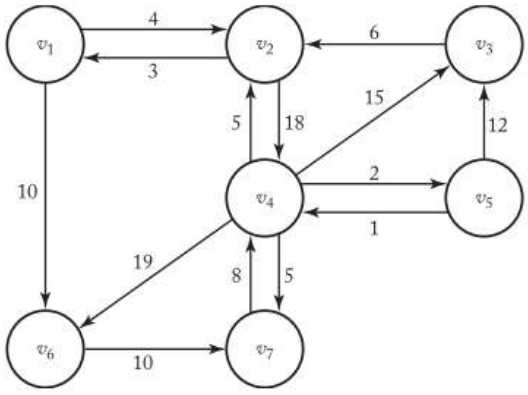
4) Modify Algorithm 3.2 (Binomial Coefficient Using Dynamic Programming) so that it uses only a one-dimensional array indexed from 0 to k.

```
int bin3 (int n, int k) {
        index i ;
        int B[0..k ] = {0};                    //initialize with zeroes
        int saved_old,  saved_new;             //two temporary variables
        for (i = 0; i <= n; i++) {
                saved_old = saved_new = B[0];
                for (j = 0;  j <= minimum(i ,k);  j++)
                        if (j == 0 || j == i)
                                B[ j ] = 1;
                else {
                        saved_new = B[j];
                        B[ j ] = saved_old + B[j ];
```

```
                    saved_old = saved_new;
                } //end else
            } //end for
            return B[k ];
        }
```

Section 3.2

5) Use Floyd's algorithm for the Shortest Paths problem 2 (Algorithm 3.4) to construct the matrixD, which contains the lengths of the shortest paths, and the matrixP, which contains the highest indices of the intermediate vertices on the shortest paths, for the following graph. Show the actions step by step.



Constructing D:

D_0:

0	4	INF	INF	INF	10	INF
3	0	INF	18	INF	INF	INF
INF	6	0	INF	INF	INF	INF
INF	5	15	0	2	19	5
INF	INF	12	1	0	INF	INF
INF	INF	INF	INF	INF	0	10
INF	INF	INF	8	INF	INF	0

D_1:

0	4	INF	INF	INF	10	INF

3	0	INF	18	INF	13	INF
INF	6	0	INF	INF	INF	INF
INF	5	15	0	2	19	5
INF	INF	12	1	0	INF	INF
INF	INF	INF	INF	INF	0	10
INF	INF	INF	8	INF	INF	0

D_2:

0	4	INF	22	INF	10	INF
3	0	INF	18	INF	13	INF
9	6	0	24	INF	19	INF
8	5	15	0	2	18	5
INF	INF	12	1	0	INF	INF
INF	INF	INF	INF	INF	0	10
INF	INF	INF	8	INF	INF	0

D_3:

0	4	INF	22	INF	10	INF
3	0	INF	18	INF	13	INF
9	6	0	24	INF	19	INF
8	5	15	0	2	18	5
21	18	12	1	0	31	INF
INF	INF	INF	INF	INF	0	10
INF	INF	INF	8	INF	INF	0

D_4:

0	4	37	22	24	10	27
3	0	33	18	20	13	23
9	6	0	24	26	19	29
8	5	15	0	2	18	5
9	6	12	1	0	19	6
INF	INF	INF	INF	INF	0	10
16	13	23	8	10	26	0

D_5:

0	4	36	22	24	10	27
3	0	32	18	20	13	23
9	6	0	24	26	19	29
8	5	14	0	2	18	5
9	6	12	1	0	19	6
INF	INF	INF	INF	INF	0	10
16	13	22	8	10	26	0

D_6:

0	4	36	22	24	10	20
3	0	32	18	20	13	23

9	6	0	24	26	19	29
8	5	14	0	2	18	5
9	6	12	1	0	19	6
INF	INF	INF	INF	INF	0	10
16	13	22	8	10	26	0

D_7:

0	4	36	22	24	10	20
3	0	32	18	20	13	23
9	6	0	24	26	19	29
8	5	14	0	2	18	5
9	6	12	1	0	19	6
26	23	32	18	20	0	10
16	13	22	8	10	26	0

Constructing P:

P_0:

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

P_1:

0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

P_2:

0	0	0	2	0	0	0
0	0	0	0	0	0	0
2	0	0	2	0	2	0
2	0	0	0	0	2	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

P_3:

0	0	0	2	0	0	0
0	0	0	0	0	0	0

2	0	0	2	0	2	0
2	0	0	0	0	2	0
3	3	0	0	0	3	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0

P_4:

0	0	4	2	4	0	4
0	0	4	0	4	0	4
2	0	0	2	4	2	4
2	0	0	0	0	2	0
4	4	0	0	0	4	4
0	0	0	0	0	0	0
4	4	4	0	4	4	0

P_5:

0	0	5	2	4	0	4
0	0	5	0	4	0	4
2	0	0	2	4	2	4
2	0	5	0	0	2	0
4	4	0	0	0	4	4
0	0	0	0	0	0	0
4	4	5	0	4	4	0

P_6:

0	0	5	2	4	0	6
0	0	5	0	4	0	4
2	0	0	2	4	2	4
2	0	5	0	0	2	0
4	4	0	0	0	4	4
0	0	0	0	0	0	0
4	4	5	0	4	4	0

P_7:

0	0	5	2	4	0	6
0	0	5	0	4	0	4
2	0	0	2	4	2	4
2	0	5	0	0	2	0
4	4	0	0	0	4	4
7	7	7	7	7	0	0
4	4	5	0	4	4	0

6) Use the Print Shortest Path algorithm (Algorithm 3.5) to find the shortest path from vertex v7 to vertex v3, in the graph of Exercise 5, using the matrix P found in that exercise. Show the actions step by step.

The shortest path from v_7 to v_3 is: v_7 -> v_4 -> v_5 -> v_3

Steps:

1) Given P7 from Exercise 5

P_7:

0	0	5	2	4	0	6
0	0	5	0	4	0	4
2	0	0	2	4	2	4
2	0	5	0	0	2	0
4	4	0	0	0	4	4
7	7	7	7	7	0	0
4	4	5	0	4	4	0

and q=7, r=3, call path(7, 3).

2) Call path(7, 5).

3) Call path(7, 4) -> P[q][r]==0, so return.

4) Print v_4.

5) Call path(4, 5) -> P[q][r]==0 so return.

6) Return from call to path(7, 5).

7) Print v_5.

8) Call path(4 ,5) -> P[q][r] ==0, so return.

9) Return from call to path(7, 3).

10) Done. Intermediate vertices v_4 and v_5 on shortest path from v_7 to v_3 have been printed in order.

7) Analyze the Print Shortest Path algorithm (Algorithm 3.5) and show that it has a linear-time complexity.

To evaluate complexity, we count the number of array accesses P[i][j].
The path from any vertex q to any vertex r has in the worst case n-1 edges (n is the nr. of vertices in the graph). For each of the n-1 edges, Algorithm 3.4 evaluates P[i][j] four times, so the worst-case complexity is $4(n-1) \in \Theta(n)$.

8) Implement Floyd's algorithm for the Shortest Paths problem 2 (Algorithm 3.4) on your system, and study its performance using different graphs.

Implementations and performance will vary.

9) Can Floyd's algorithm for the Shortest Paths problem 2 (Algorithm 3.4) be modified to give just the shortest path from a given vertex to another specified vertex in a graph? Justify your answer.

Working backwards from $D^{(n)}(i,j)$, we note that it only depends on 3 values from the previous table $D^{(n-1)}$. Based on this it is possible to reduce the number of operations by only calculating one value from $D^{(n)}$, three values from $D^{(n-1)}$, etc. As we regress towards $D^{(0)}$, however, the number of values grows and ends up being n^2 (the entire table), and the asymptotic complexity remains $\Theta(n^3)$.

10) Can Floyd's algorithm for the Shortest Paths problem 2 (Algorithm 3.4) be used to find the shortest paths in a graph with some negative weights? Justify your answer.

Yes, as long as there are no cycles whose total length is negative (in which case the path can be made arbitrarily small by repeating such negative cycles).

Section 3.3

11) Find an optimization problem in which the principle of optimality does not apply and therefore the optimal solution cannot be obtained using dynamic programming. Justify your answer.

The text example: Finding the <u>longest path</u> between two vertices A → C of a graph (without loops) does not have optimal substructure: if the longest path is A → … → B → … → C, this <u>doesn't</u> mean that:

- the subpath B → … → C is longest; for instance, in the B→C problem we may find that the longest path contains A, which would create a cycle in the original problem A → C, so it's not allowed there.
- the subpath A → … → B is longest; for instance, in the A→B problem we may find that the longest path contains C.

Another example: Raising a number to an integer power n with the fewest number of multiplications (a.k.a. the Addition Chain Exponentiation problem). When n is a power of 2, the best way is successive squaring, e.g. $x^8 = ((x \cdot x)^2)^2$ requires only 3 multiplications. If n is not a power of 2, however, there is no optimal substructure, because a subproblem is not always solved optimally by itself; we can prefer a suboptimal solution to a subproblem because parts

of the solution can be reused to solve other subproblems. This problem was proved to be NP-complete (see Ch.9).

Section 3.4

12) List all of the different orders in which we can multiply five matrices A, B, C, D, and E.

There are 14 possible orderings:

A(B((CD)E))

A(B(C(DE)))

A((BC)(DE))

A(((BC)D)E)

A((B(CD))E)

(AB)((CD)E)

(AB)(C(DE))

(A(BC))(DE)

((AB)C)(DE)

(A(B(CD)))E

(A((BC)D))E

((AB)(CD))E

((A(BC))D)E

(((AB)C)D)E

13) Find the optimal order, and its cost, for evaluating the product A1×A2× A3×A4×A5, where

A1 is (10×4)
A2 is (4×5)
A3 is (5×20)
A4 is (20×2)
A5 is (2×50)

Show the final matrices M and P produced by Algorithm 3.6.

M:

0	200	1200	320	**1320**
	0	400	240	640
		0	200	240
			0	2000
				0

P:

	1	1	1	4
		2	2	4
			3	4
				4

Optimal order = (A1(A2(A3·A4)))A5 Optimal cost: 1320 multiplications

14) Implement the Minimum Multiplications algorithm (Algorithm 3.6) and the Print Optimal Order algorithm (Algorithm 3.7) on your system, and study their performances using different problem instances.

Implementations and performances will vary.

15) Show that a divide-and-conquer algorithm based on Equality 3.5 has an exponential-time complexity.

$$M[i][j] = \underset{i \le k \le j-1}{minimum}(M[i][k] + M[k+1][j] + d_{i-1}d_k d_j), \text{ if } i < j.$$
$$M[i][i] = 0. \qquad (3.5)$$

For illustration, we use the M table from Exercise 13:

0	200	1200	320	**1320**
	0	400	240	640
		0	200	240
			0	2000
				0

As in the analysis of the Minimum Multiplications algorithm, we take the basic operation to be one evaluation of k. A problem of size n (obtaining the 1320 in example above) requires evaluation of more than two problems of size n-1 (320 and 640 above), so we have the inequality $T(n) \ge 2T(n-1) \ge 2 \cdot 2T(n-2) \ge \ldots \ge 2^n$.

16) a) Problem: Determine the number of different orders in which n matrices can be multiplied.

Input: An integer n representing the number of matrices to be multiplied.

Output: Number of different orders in which n matrices can be multiplied.

```
int numOrderings(int n) {
        rec(1, n);
}
int recNumOrderings(int j, int k) {
        if(k-j<=1) return 1;
        int sum = 0;
         for(int i=j; i<k; i++) {
                sum += rec(j,i)*rec(i+1,k);
         }
         return sum;
}
```

b) n=2: 1

n=3: 2

n=4: 5

n=5: 14

n=6: 42

n=7: 132

n=8: 429

n=9: 1430

n=10: 4862

17) Establish the equality

$$\sum_{diagonal=1}^{n-1} [(n - diagonal) \times diagonal] = \frac{n\,(n-1)\,(n+1)}{6}.$$

By multiplying out each term inside the sum, we have: $n\sum_{d=1}^{n-1} d - \sum_{d=1}^{n-1} d^2$. The first sum is solved in Example A.1, and the second in Example A.2; we only have to replace n with n-1:

$$n\frac{(n-1)n}{2} - \frac{(n-1)n[2(n-1)+1]}{6} = \frac{n(n-1)(n+1)}{6}.$$

18) Show that to fully parenthesize an expression having n matrices we need n−1 pairs of parentheses.

Induction on n:
Base case: for n = 1, we have only one matrix, which needs 0 pairs. n – 1 = 1 – 1 = 0.
Hypothesis: Result holds up to n.
Induction step: We show that the result holds for n + 1: Number the matrices from 1 to n, and let k be the "breaking point" of the top-level product:

$$(A_1 \ldots A_k)\cdot(A_{k+1} \ldots A_{n+1})$$

The left factor has k matrices, and the right has n-k+1. Both numbers are ≤ n, so we can apply to them the hypothesis: the left needs k – 1 pairs, and the right n – k, for a total of n – 1. We now apply the outer parenthesis to obtain n pairs:

$$((A_1 \ldots A_k)\cdot(A_{k+1} \ldots A_{n+1})).$$

19) Analyze Algorithm 3.7 and show that it has a linear-time complexity.

Every call to *order* prints one pair of parentheses, and according to Exercise 18, a problem of size n needs only n-1 pairs, which is $\Theta(n)$.

20) Write an efficient algorithm that will find an optimal order for multiplying n matrices A1 ×A2×⋯×An, where the dimension of each matrix is 1×1, 1×d, d×1, or d×d for some positive integer d. Analyze your algorithm and show the results using order notation.

We use the array of (n+1) dimensions d from Algorithm 3.6, but here we call it Dim to distinguish it from the integer d. The elements of Dim are either 1 and d.

1. We scan Dim and make each 1 the boundary of a sub-product. ($\Theta(n)$)

2. For each sub-product $A_i \ldots A_j$:

- If both A_i … A_j: are 1x1, then we can multiply all the intervening dxd matrices in any order. We arbitrarily choose $A_{i+1}(A_{i+2}(\ldots A_{j-1}) \ldots)$ $(\Theta(j-i))$
- If A_i is 1xd, then we multiply from left to right $A_i(A_{i+1}(\ldots A_j) \ldots)$ $(\Theta(j-i))$
- If A_j is dx1, then we multiply from right to left $(\ldots(A_i(\ldots A_{j-1})A_j)$ $(\Theta(j-i))$

All the branches of Step 2 are linear, with the same multiplicative constant (the numbers of operations needed to insert a pair of parentheses), so the entire Step 2 is $\Theta(n)$, which makes the entire algorithm $\Theta(n)$.

Section 3.5

21) How many different binary search trees can be constructed using six distinct keys?

By the formula in Exercise 36: (1/7)*(12 choose 6) = (1/7)*924 = 132

22) Create the optimal binary search tree for the following items, where the probability occurrence of each word is given in parentheses: CASE (.05), ELSE (.15), END (.05), IF (.35), OF (.05), THEN (.35).

For simplicity, number the words from CASE = 1 to THEN = 6

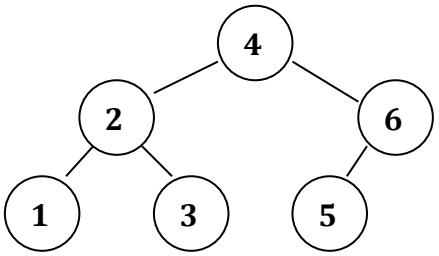
A:

	0	**1**	**2**	**3**	**4**	**5**	**6**
1	0	.05	.25	.35	.95	1.05	1.8
2		0	.15	.25	.8	.9	1.65
3			0	.05	.45	.55	1.3
4				0	.35	.45	1.2
5					0	.05	.45
6						0	.35
7							0

R:

	0	**1**	**2**	**3**	**4**	**5**	**6**
1	0	1	2	2	4	4	4
2		0	2	2	4	4	4
3			0	3	4	4	4
4				0	4	4	4
5					0	5	6
6						0	6
7							0

Optimal tree:



23) Find an efficient way to compute $\sum_{m=i}^{j} p_m$ which is used in the Optimal Binary Search Tree Algorithm (Algorithm 3.9).

The partial sums can be stored in an n x n table B, whose position B[i][j] stores $\sum_{m=i}^{j} p_m$. We have the recursion B[i][j+1] = B[i][j] + p_{j+1}.

Below is a partial example with the probabilities from Exercise 22:

	1	**2**	**3**	**4**	**5**	**6**
1	.05 →	.20 →	.25			
2		.15 ↑	↑			
3			.05			
4				.35		
5					.05	
6						.35

24) Implement the Optimal Binary Search Tree algorithm (Algorithm 3.9) and the Build Optimal Binary Search Tree algorithm (Algorithm 3.10) on your system, and study their performances using different problem instances.

Implementations and performances will vary.

25) Analyze Algorithm 3.10, and show its time complexity using order notation.

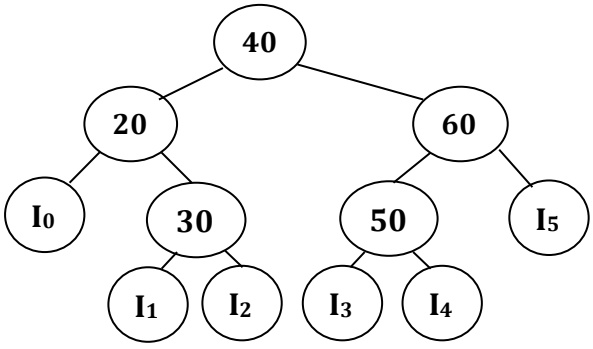
We define the basic operation to be the access of an element of the array R; this is done once per each call to the function *tree*, and each such call inserts in the tree one node. Since there are n nodes total, the number of calls and that of basic operations is n, so the complexity is linear, $\Theta(n)$.

26) Generalize the Optimal Binary Search Tree algorithm (Algorithm 3.9) to the case in which the search key may not be in the tree. That is, you should let q_i, in which i =0, 1, 2, . . . , n, be the probability that a missing search key can be situated between Key_i and Key_{i+1}. Analyze your generalized algorithm and show the results using order notation.

The solution is very similar, only, aside from the keys, we now also have the *intervals*

$I_0 = [-\infty \ldots Key_1]$, $I_1=[Key_1 \ldots Key_2]$, … , $I_{n-1} = [Key_{n-1} \ldots Key_n]$, $I_n = [Key_n \ldots +\infty]$

The intervals can be thought of as leaves of the tree,



however, they are not really there, i.e. there are no pointers pointing to them. Their probabilities simply add up into the sum of probabilities of the keys: instead of $\sum_{m=i}^{j} p_m$, we now have $\sum_{m=i}^{j} p_m + \sum_{m=i-1}^{j} q_m$. The search function can either stop at an internal node (key), or "at a

leaf" … but the leaves are not accessed through an extra pointer operation from the key, so the depth of the interval represented by the leaf is the same as its parent key.

Complexity is identical: $\Theta(n^3)$.

27) Show that a divide-and-conquer algorithm based on Equality 3.6 has an exponential time complexity.

For illustration, we use the A table from Exercise 22:

	0	**1**	**2**	**3**	**4**	**5**	**6**
1	0	.05	.25	.35	.95	1.05	1.8
2		0	.15	.25	.8	.9	1.65
3			0	.05	.45	.55	1.3
4				0	.35	.45	1.2
5					0	.05	.45
6						0	.35
7							0

As in the analysis of the Minimum Multiplications algorithm, we take the basic operation to be one evaluation of k. A problem of size n (obtaining the 1.8 in example above) requires evaluation of more than two sub-problems of size n-1 (1.05 and 1.65 above), so we have the inequality $T(n) \geq 2T(n-1) \geq 2 \cdot 2T(n-2) \geq \ldots \geq 2^n$.

Section 3.6

28) Find an optimal circuit for the weighted, direct graph represented by the following matrix W. Show the actions step by step.

$$W = \begin{bmatrix} 0 & 8 & 13 & 18 & 20 \\ 3 & 0 & 7 & 8 & 10 \\ 4 & 11 & 0 & 10 & 7 \\ 6 & 6 & 7 & 0 & 11 \\ 10 & 6 & 2 & 1 & 0 \end{bmatrix}$$

D[v2][Ø] = 3 D[v3][Ø] = 4 D[v4][Ø] = 6 D[v5][Ø] = 10

D[v2][v3] = 11 D[v2][v4] = 14 D[v2][v5] = 20

D[v3][v2] = 14 D[v3][v4] = 16 D[v3][v5] = 17

D[v4][v2] = 9 D[v4][v3] = 11 D[v4][v5] = 21

D[v5][v2] = 9 D[v5][v3] = 6 D[v5][v4] = 7

D[v2][v3,v4] = max(7+16, 8+11) = 19, j = 3 D[2][3,5] = 16, j = 5 D[2][4,5] = 16, j = 5

D[3][2,4] = 19, j = 3 D[3][2,5] = 16, j = 5 D[3][4,5] = 14, j = 5

D[4][2,3] = 17, j = 2 D[4][2,5] = 20, j = 5 D[4][3,5] = 17, j = 5

D[5][2,3] = 16, j = 3 D[5][2,4] = 10, j = 4 D[5][3,4] = 12, j = 4

D[2][3,4,5] = 22, j=5 D[3][2,4,5] = 17, j=5 D[4][2,3,5] = 22, j=2 D[5][2,3,4] = 18, j=4

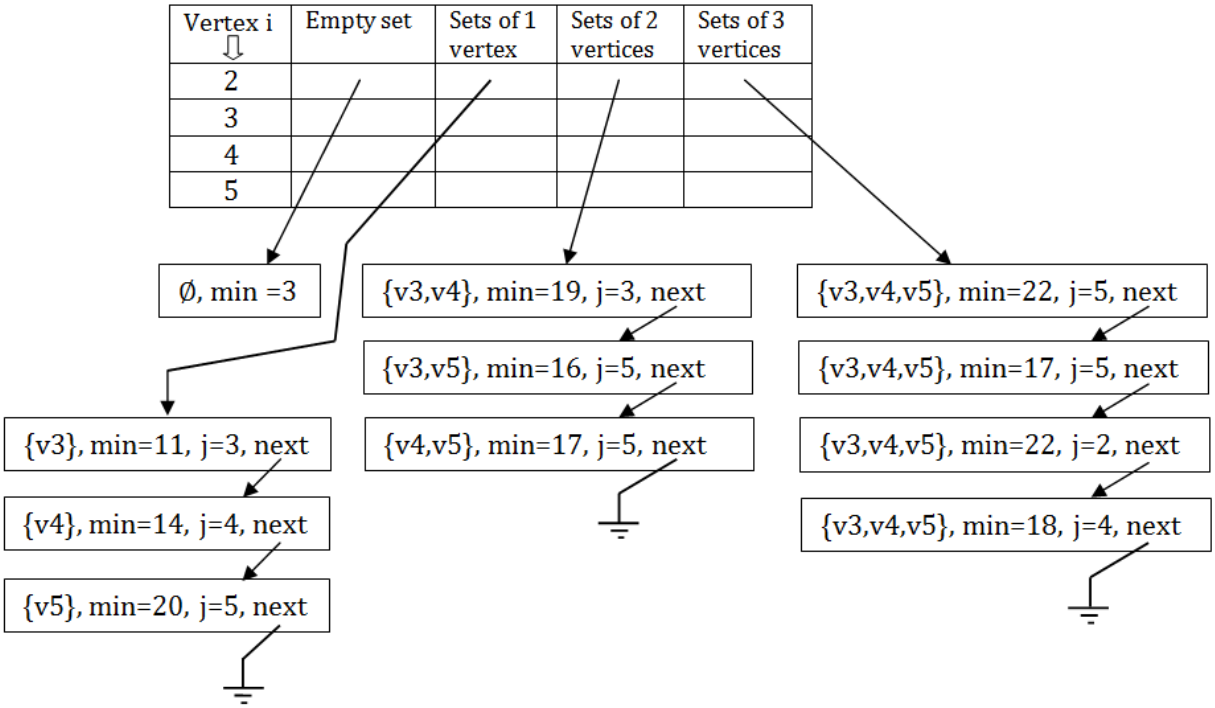
D[1][2,3,4,5] = min(30, 30, 40, 38), which shows that there are two minimal tours of cost 30, one with j = 2, the other with j = 3:

Minimal Tour A: 1→2→5→4→3→1 Minimal Tour B: 1→3→5→4→2→1

29) Write a more detailed version of the Dynamic Programming algorithm for the TSP (Algorithm 3.11).

One design decision is how to store the subsets of vertices. This can be done, for example, with an array of n linked lists, each of them holding the set of vertices (from zero – empty set – to (n – 2) sets) and the respective optimal cost and j vertex associated.

Below is an example of matrix D for n = 5 (from Exercise 28). The "ground" symbol means a null pointer. Only the line for v2 is shown:



The sets themselves can be stored as strings, arrays of boolean (used below), or even the data structure described in Appendix C.

For homogeneity, the list node for the empty set can be made the same as the others (although technically it does not need a j index or a next pointer):

```
struct NodeType{
        bool set[n-1];
        int min;
        int jay;
        struct NodeType* next;
};
```

When we generate the subsets $A \subseteq V - \{v1\}$ containing k vertices, we need to store them in an array:

bool arrayOfSubsets[1..maxSubsets][n-1]

where maxSubsets is the maximum over k of Binomial(n-2, k), so it can store the most subsets that can be generated. Alternatively, the subsets can also be stored in a linked list.

The more detailed pseudocode is:

```
void travel (int n, const number W[][], index P[][], number& minlength) {
    index i , j , k, temp_j, minIndex;
    number value, temp_min;
    struct nodeType* D[2..n][ 0..n-2];
    for (i = 2; i <= n; i++)
        D[ i ][0]->min = W[ i ][1];
    for (k = 1; k <= n−2; k++) {
```

```
            Generate all subsets A ⊆ V−{v1} containing k vertices, store them in arrayOfSubsets;
            for ( all subsets A in arrayOfSubsets)
                  for (i such that i != 1 and v_i is not in A) {
                        temp_min = INFINITY;
                        for (each j:v_j∈A) {
                              find in the list D[ j ][k-1] the node x containing the set A− {v_j};
                              value = W[ i ][ j ] + x.min;
                              if (value < temp_min) {
                                    temp_min = value;
                                    temp_j = j ;
                              }
                        }
                        Navigate to end of list D[i][k];
                        Append a node y with:
                              y.min =  temp_min;
                              y.jay = temp_j;
                  } //end for i
            }
      } //end for k
      minlength = minimum (W[1][ j ] + D[ j ][V− {v1,vj}]);
                  2≤j≤n
      minIndex = value of j that gave the minimum;
}
```

30) Implement your detailed version of Algorithm 3.11 from Exercise 27 on your system and study its performance using several problem instances.

Implementations and performances will vary.

Section 3.7

31) Analyze the time complexity of Algorithm *opt*, which appears in Section 3.7.

The complexity of the recursive version (Algorithm 3.12) is worse than exponential:

Denote by T(n,m) the complexity of calculating *opt*(0,0) for the original two sequences. Without loss of generality, let n be the smaller length. Clearly, $T(n,m) \le T(n,n)$, which in turn we denote S(n). The recursive call on the last line of Algorithm 3.12 shows that each instance of *opt* calls three sub-instances with sizes n-1 or larger, so we have $S(n) \le 3S(n-1)$, which by repeated substitution leads to $S(n) \le 3^n$.

If we use the Dynamic Programming version, however, the table used in the solution has dimensions [0..n+1] and [0..m+1], therefore there are (n+1)(m+2) locations i, j at which *opt* must be called. Each call reads at most three locations that have already been filled in, so the complexity is $\leq 3(n+2)(m+2) \in \Theta(nm)$.

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

Input: A DNA sequence x of length m and a DNA sequence y of length n represented as arrays.

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

```
int align(int X[], int Y[]) {
index i, j;
    int opt[0…m][0…n];
    for(i=0; i<=n; i++)
            opt[i][n] = 2*(m-i);
    for(j=0; j<=m; j++)
            opt[m][j] = 2*(n-j);
    for(i=m-1; i>=0; i--) {
        for(j=n-1; j>=0; j--)  {
            if(X[i] == Y[j]) penalty = 0;
            else penalty  = 1;
            opt[i][j] = min(opt[i+1][j+1] + penalty, opt[i+1][j] + 2, opt[i][j+1] + 2);
        }
    }
    return opt[0][0];
}
```

33)

	0 G	1 G	2 A	3 G	4 T	5 T	6 C	7 A	8 -
0 C	8	9	11	12	14	16	18	20	22
1 C	6	7	9	10	12	14	16	18	20
2 G	5	5	7	8	10	12	14	16	18
3 G	4	5	5	6	8	10	12	14	16

4 G	6	4	5	4	6	8	10	12	14
5 T	7	6	4	4	4	6	8	10	12
6 T	8	6	5	3	3	4	6	8	10
7 A	9	7	5	4	2	3	4	6	8
8 C	11	9	7	5	3	1	2	4	6
9 C	12	10	8	6	4	2	0	2	4
10 A	14	12	10	8	6	4	2	0	2
11 -	16	14	12	10	8	6	4	2	0

Optimal alignment is CCGGGTTACCA with –GGAGTT-C-A, which has cost 8.

34) Like algorithms for computing the nth Fibonacci term (see Exercise 42 in Chapter 1), the input size in Algorithm 3.2 (Binomial Coefficient Using Dynamic Programming) is the number of symbols it takes to encode the numbers n and k. Analyze the algorithm in terms of its input size.

The complexity of Algorithm 3.2 in terms of n and k is $\Theta(nk)$, and the maximum k is n, so in the worst case $\Theta(n^2)$. If the input is encoded in binary, we have lg n symbols for n and maximum lg n symbols for k = n, for a total of 2lg n = lg n^2. Denote n^2 by m → the input is lg m, and the ouput is $\Theta(m)$. Now denote lg m by p → The complexity is $m = 2^p$, so the complexity is exponential in terms of the nr. of symbols (bits) in the input.

35) Determine the number of possible orders for multiplying n matrices A1, A2, . . . , An.

The number of ways of fully parenthesizing n factors is given by the $(n-1)^{st}$ Catalan number C_{n-1} = (1/n)*((2*(n-1)) choose (n-1)).

36) Show that the number of binary search trees with n keys is given by the formula

$$\frac{1}{(n+1)}\binom{2n}{n}.$$

Denote by B(n) the number of BSTs with n keys. Any of the keys can be the root. If key j is the root, then keys 1 … j-1 must be in the left subtree, and keys j+1…n in the right subtree. The left

and right subtrees are in turn BSTs, with (j-1) and (n-j) keys, respectively, so we have the recursion:

$$B(n) = \sum_{j=1}^{n} B(j-1)B(n-j), \quad \text{where } B(0) = 1 \qquad (*)$$

For instance:

- for B(1) the recursion yields B(0)B(0) = 1·1 = 1, which is correct.
- for B(2): B(0)B(1) + B(1)B(0) = 1 + 1 = 2, also correct.

It can be proved that the number given $\frac{1}{(n+1)}\binom{2n}{n}$, a.k.a. the n^{th} Catalan number satisfies the recursion (*).

37) Can you develop a quadratic-time algorithm for the Optimal Binary Search Tree problem (Algorithm 3.9)?

Inside the innermost *for* loop of Algorithm 3.9 we have the calculation of A[i][j], which requires a minimum for values of k ranging from i to j:

$$\begin{aligned} &j = i + diagonal; \\ &A[i][j] = \underset{i \le k \le j}{minimum}(A[i][k-1] + A[k+1][j]) + \sum_{m=i}^{j} p_m . \\ &R[i][j] = \text{a value of } k \text{ that gave the minimum;} \end{aligned}$$

In the paper by Donald E. Knuth, "Optimum binary search trees", Acta Informatica 1 (1): 14–25, (1971), it is proved that R[i][j] is always in between R[i][j-1] and R[i+1][j]. Below is an example that uses the second diagonal of matrix R from Exercise 22:

	0	**1**	**2**	**3**	**4**	**5**	**6**
1	0	1	2	2	4	4	4
2		0	2	2	4	4	4
3			0	3	4	4	4
4				0	4	4	4
5					0	5	6
6						0	6

We replace in the algorithm $\underset{i \le k \le j}{minimum}$ with $\underset{R[i][j-1] \le k \le R[i+1][j]}{minimum}$.

The values of k we now need to examine for an entire diagonal are at most twice the range of the previous diagonal, which is at most 1 through n. There are n diagonals, so the complexity is $T(n) \le 2n^2 \in O(n^2)$.

38) Problem: Find the maximum sum in any contiguous sub-list in a list of n numbers.

Input: A list S[] of n numbers indexed from 1 to n.

Output: The maximum sum in any contiguous sub-list of S[].

```
number maxSublist(int n, number S[]) {
    index i=1;
    number minTotal = min(0, S[1]);
    number total = S[1];
    number val = S[1];
    for(i=2; i<=n; i++) {
            total = total + S[i];
            if(total – minTotal > val)
                    val = total – minTotal;
            if(total < minTotal)
                    minTotal = total;
    }
    return val;
}
```

The algorithm performs exactly n iterations for a list of n numbers, each of which contains a constant number of operations, and so its complexity is O(n).

39) Problem: Find the longest common subsequence of two sequences of characters S1 and S2.

Input: Two sequences of characters S1 and S2 and integers m and n representing their respective lengths.

Output: An array C[0…m][0…n] whose element C[m][n] contains the length of the longest common subsequence.

```
int[][]  LCS(char S1[], int m, char S2,[] int n) {
    int C[0…m][0…n];
index i, j;
    for(i=0; i<=m; i++)
            C[i][0] = 0;
```

```
    for(j=0; j<=n; j++)
            C[0][j] = 0;
    for(i=1; i<=m; i++) {
            for(j=0; j<=n; j++) {
                    if(S1[i]==S2[j])
                            C[i][j] = C[i-1][j-1];
                    else
                            C[i][j] = max(C[i][j-1], C[i-1][j]);
            }
    }
    return C;
}
 function  LCSLength(X[1..m], Y[1..n])
  C = array(0..m, 0..n)
 for i := 0..m
     C[i,0] = 0
    for j := 0..n
         C[0,j] = 0
        for i := 1..m
         for j := 1..n
        if X[i] = Y[j]
            C[i,j] := C[i-1,j-1] + 1
       Else:
             C[i,j] := max(C[i,j-1], C[i-1,j])
      return C[m,n]
```

The longest common subsequence can be printed out by calling the following function using the C array resulting from running the LCS function above, the two original sequences S1 and S2, and the lengths of the sequences m and n.

```
void printLCS(int C[][], char S1[], char S2[], int i, int j) {
    if(i==0 or j==0)
            return "";
    if(S1[i] == S2[i])
```

```
            return printLCS(C, S1, S2, i-1, j-1) + S1[i];
        if(C[i, j-1] > C[i-1, j])
            return printLCS(C, S1, S2, i, j-1);
        return printLCS(C, S1, S2, i-1, j);
    }
```